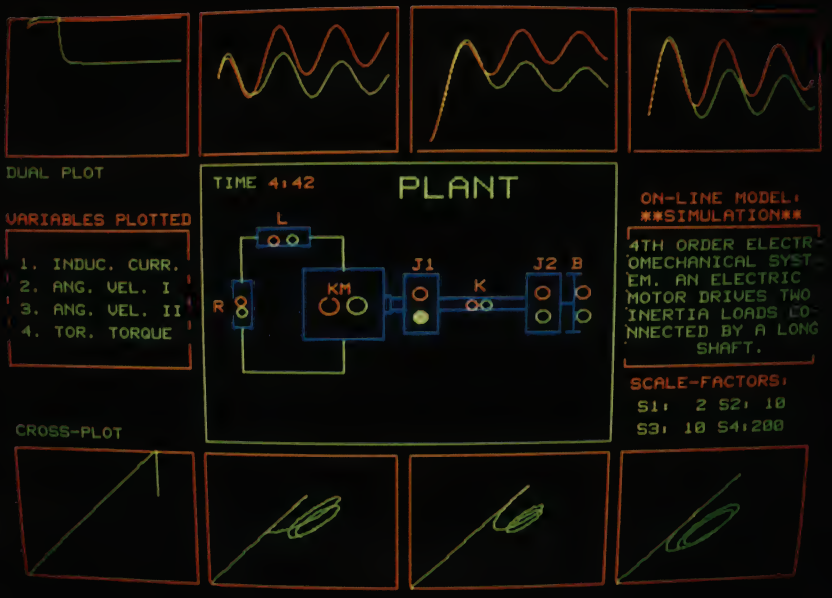
VAXELN Technical Summary



digital

VAXELN Technical Summary

digital

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The following are trademarks of Digital Equipment Corporation: the Digital logo, DEC, PDP, VAX, VT, Rainbow, DECmate, Professional, ALL-IN-1, DECnet, VAXELN, MicroVAX, and MicroVMS.

Printed in U.S.A. Copyright © 1984 Digital Equipment Corporation
All Rights Reserved

Table of Contents

1 • VAXELN: Introduction and Overview 1

2 • The Kernel. 7

3 • Language: Powerful Programming in PASCAL. 22

4 • Developing a VAXELN Application. 27

5 • Communication: Transparent Networks 34

6 • VAX Hardware: Systems of Choice 40

7 • Service and Support. 51

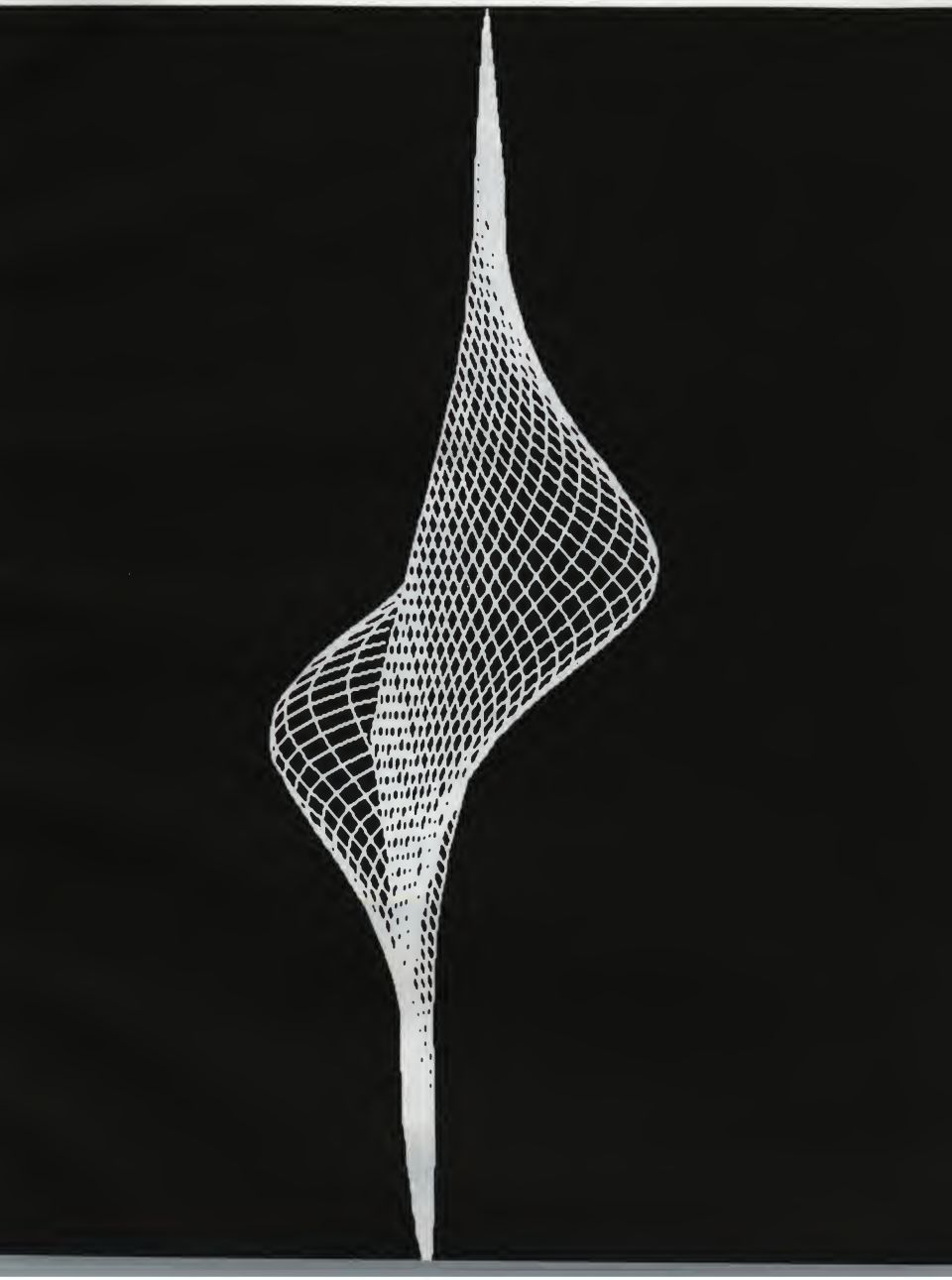
Appendix I 56

Appendix II. 62

Appendix III 72

Index 85

Chapter 1· VAXELN: Introduction and Overview.



What Is VAXELN?

VAXELN realtime programming software is a powerful and sophisticated tool kit for creating specialized VAX software systems. It is a VMS optional product that runs on VAX superminicomputers and MicroVAX microcomputers and supports the development of stand-alone software systems.

VAXELN systems include only those services (user programs or Digital-supplied programs) that are needed to support the functions of the system. In other words, systems are only as complex as they need be to do their work. We call such systems "statically defined."

Optimized for execution speed and efficiency, VAXELN programs are perfect for areas where general purpose systems just won't do. This includes demanding realtime environments, dedicated distributed processing, and other tasks in which processors have predetermined functions and are not needed simultaneously for general computing, program development, or other uses. Industrial automation, workstations, and Ethernet server networks are typical VAXELN applications.

Although applications are by no means limited to realtime or networking jobs, VAXELN systems are most formidable in these areas. Especially suited to creating realtime applications, VAXELN software simplifies the design and implementation of such systems. The conceptually simple, small object-oriented kernel executive supplies basic runtime services efficiently. Optional service programs and device drivers implement a file system, network communication facilities, and I/O device handling, with minimum overhead.

It's an Optional Product on VMS.
The VAXELN Toolkit is an optional product that runs on a VAX processor under the VMS operating system, Version 3.5 or higher, or under MicroVMS Versions 1.0 and 4.0. The tool kit consists of a compiler, System Builder utility, kernel executive, and debugger. During program development, several standard VMS utilities are used like the VMS linker and editor utilities and the library facility.

VAXELN applications run on VAX minicomputers and MicroVAX microcomputers, with no general purpose operating system present. The development system is called the VAXELN host processor: the execution system is called the target processor. Your finished VAXELN system can be loaded onto portable Files-11 B storage volumes and booted from them on the target computer. Or, if you have the optional DECnet-VAX license and Ethernet hardware, they can be loaded downline into the target computer. Also, you can blast your application into read only memory for installation into the target system.

The VAXELN System at a Glance.
A VAXELN system is a set of programs executing on VAX hardware, along with standard code and data that manage the programs' execution.

You develop a new VAXELN system by writing whatever new programs are required in VAXELN PASCAL or using existing programs written in other VAX/VMS languages. Then, with the tool kit's System Builder utility, you combine the programs—with each other, with any of the drivers and services you want, and with the VAXELN kernel—to form a functioning system.

VAXELN can combine one or more VAX or MicroVAX processors, optional peripheral devices (including disks and terminals), and communication hardware to support the execution of the programs in a local area network. In addition, VAXELN hardware configurations can include special hardware you have designed or bought, such as custom device interfaces.

If you are programming a set of computers linked by the Ethernet, you simply prepare a system for each computer, or node, and include the VAXELN Network Service in each node's system.

VAXELN Delivers Impressive Programming Results.
With VAXELN you can obtain impressive programming results without becoming an expert at system programming. VAXELN tools let you bring your applications to life sooner by constructing concurrently executing programs for applications requiring maximum efficiency and precise timing. And you can accomplish this without learning assembly-level programming. Instead, you can write your programs in PASCAL, an easy-to-use, high-level language.

VAXELN PASCAL is a highly optimizing compiler, suitably extended to eliminate the need for other programming languages (including assembly language) in applications of this kind. It is the primary implementation language of the VAXELN tool kit itself. The VAXELN compiler can accept any ISO-standard PASCAL program.

VAXELN PASCAL is a superset of ISO-standard PASCAL, extended for more power and versatility.

Concurrent PASCAL Makes Realtime Programming Possible.

Both multitasking and multiprogramming are supported by the VAXELN kernel and VAXELN PASCAL programming language. Multitasking means that the user can declare parts of a PASCAL program even though they are concurrently scheduled for execution with the main program. That is possible because the execution of the task is interleaved with the execution of all other tasks on the same CPU, giving the effect of concurrent execution. Multiprogramming means that entire programs, including multitasking programs, can be scheduled concurrently on the same CPU.

Convenience Features Make Application Development Quicker and Easier.

VAXELN PASCAL offers several features that make programming more convenient. For instance, parts of a program can be written and compiled separately, without giving up any of the impressive typechecking features of the PASCAL language. This allows you to split up the work on a programming project, increasing productivity by assigning different programmers to work on separate sections. You can develop and debug your application code, module by module, adding each fully tested section to your total system as it is ready.

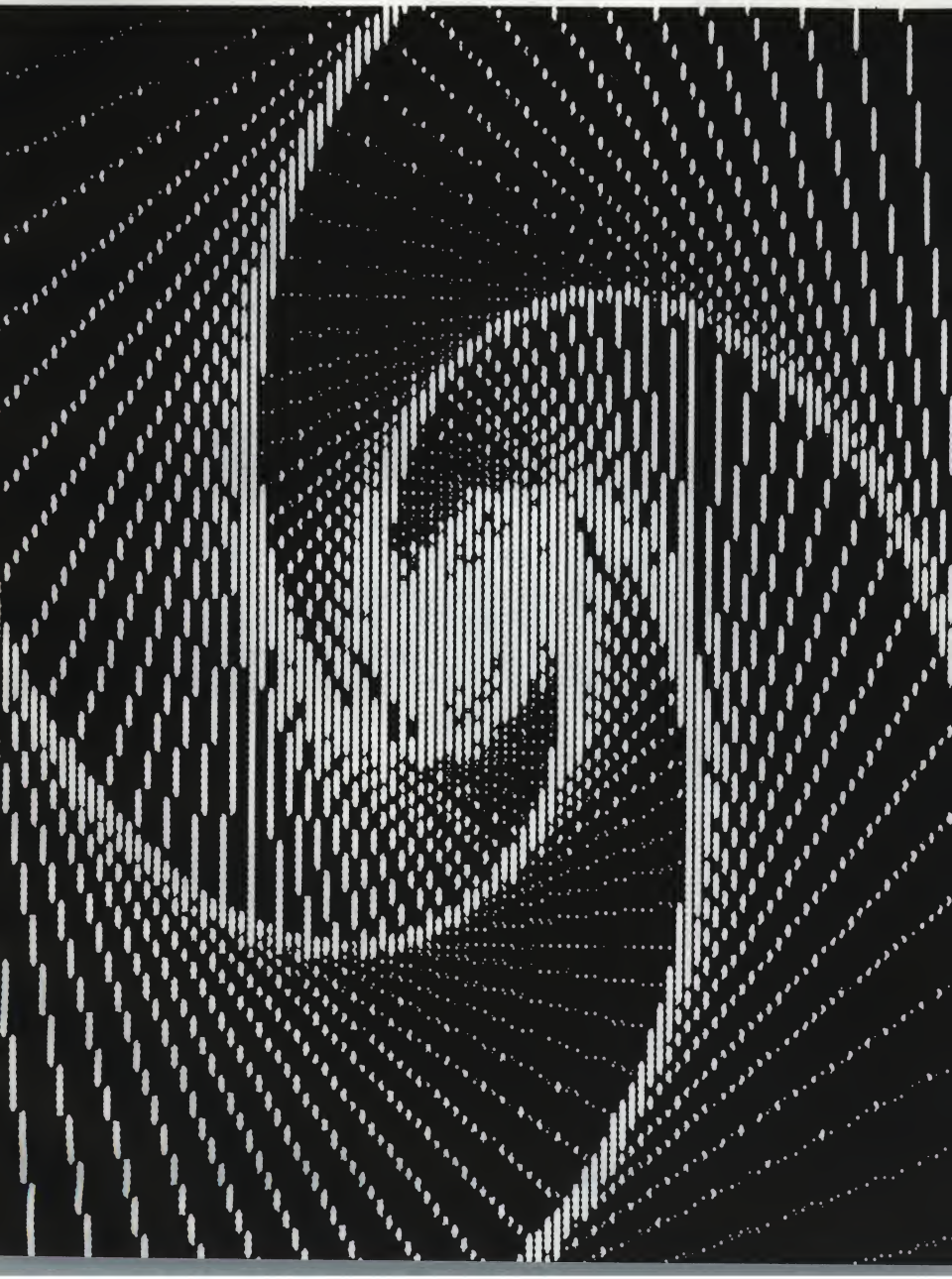
Symbolic debugging lets you examine and modify a problem program using the PASCAL symbols you wrote it in. Once again, the fact that you use a high-level language makes you more productive.

Database Management with VAX Rdb/ELN.
VAX Rdb/ELN is a relational database management system operating in a VAXELN environment.

By combining VAX Rdb/ELN with VAXELN and Ethernet you can create a fully functioning database management system that runs fast, costs less, and offers a number of unique features.

See Appendix I for more detailed information about the relationship between VAXELN and VAX Rdb/ELN, and some of the major features of VAX Rdb/ELN.

Chapter 2 · The Kernel.



The Kernel Is in Control.
The VAXELN kernel is a layer of software between the hardware and your system software. It controls the sharing of system resources and synchronizes communication among the various programs in the system.

In a simple, straightforward way, the kernel provides all the mechanisms necessary for a wide range of applications—for example, communication between processes. It also maintains all information about the system data and about the programs defined for a particular system. The kernel is delivered to you as a prepared image, ready to be built into a system along with your programs.

Services and device drivers run independently of the kernel and user programs. They can be omitted from VAXELN systems that do not need their capabilities. This, in fact, is the model of a user's VAXELN application in general: if the processes in a system or network require a complex service, it is provided as an independent program running in the system or network. For instance, a file server providing file storage hardware and software for a community of network users is built by constructing a VAXELN system from the Network Service, File Service, and disk driver and running the system on a network node that has the actual disks. VAXELN programs anywhere in the network can then use the file server without having to know its network location.

Kernel Objects
The kernel recognizes and operates on a number of objects in addition to the familiar entities (integers, strings, and so on) used in conventional programming. Each of these objects is represented in VAXELN PASCAL as a predeclared data type and has a corresponding set of operations realized in the language as procedure calls (see Appendix II).

Objects are created by CREATE procedures in VAXELN Pascal (for example, CREATE_PROCESS) and can be deleted by the generic DELETE procedure. The "IDs" of objects—the value of the system types like PROCESS are valid everywhere in the creating job (see below) and, with the exception of PORT values, only in the creating job. In general, then, VAXELN systems are systems of independent process families (jobs) that exchange information only by message transmissions.

With VAXELN the Process Is the Task.
Processes are the execution agents for programs or for concurrently scheduled parts of programs. ("Processes" compare to 'tasks" in other systems, such as RSX-11M.) the "main" program (program block) in a VAXELN PASCAL program is executed by a master process created implicitly by the kernel when the program is started. VAXELN PASCAL programs also can include special process blocks; these have the same form as a PASCAL procedure declaration, but with the word PROCESS_BLOCK in lieu of PROCEDURE.

Process blocks are executed by dynamically created subprocesses. The programmer calls the CREATE_PROCESS procedure, specifies the name of the process block, and receives a unique value of the data type PROCESS that identifies the process executing that block. (The master process, or any process, can obtain its own PROCESS value by calling another predeclared procedure.)

In VAXELN terminology, the "family" of a master process and its subprocesses is a job. Jobs can be created dynamically (with the CREATE_JOB procedure) to execute a specific program that was included with the System Builder. One copy of a program's code is shared by any number of jobs executing that program. A System

Builder option also allows the user to specify that jobs are created by the kernel, when the system is started, to run designated programs in the system and, as an additional option, to perform initialization sequences before other jobs are allowed to execute.

Jobs running on the same processor are scheduled on a preemptive priority basis with 32 levels of priority. Within a job, processes are rescheduled the same way, but with 16 levels of priority. Both process and job priorities can be changed dynamically with kernel procedures.

Processes can be explicitly removed from the rescheduling operation with the SUSPEND procedure and reinstated with the RESUME procedure. Process rescheduling within a job can be temporarily stopped and then resumed with other kernel procedure calls.

Each VAXELN process in a working system exists in one of several states: running, ready, suspended, and waiting. A high priority for a process means that it should preempt others in the same job when it is ready. Processes enter the waiting state when they call the WAIT_ANY or WAIT_ALL procedure to wait for other objects in the system. With WAIT_ANY, any of up to four object-arguments allows the caller to proceed; with WAIT_ALL, all four objects must allow the process to proceed.

For example, a process can wait for a message to arrive at a port or for an event to be signaled by the SIGNAL procedure. By waiting for the PROCESS value that identifies a different process, a process can synchronize its execution with the termination of another process.

When the wait conditions are satisfied, the waiting process enters the ready state. Resumed processes reenter the state from which they are suspended. All processes are initially ready, and the kernel continually selects the ready process with the highest priority, giving it control of the CPU.

Jobs are ready if they have at least one ready process, and the highest-priority, ready job is allowed to run. As with processes, the running job is preempted if a higher-priority job becomes ready.

The VAXELN kernel uses VAX memory management to map user processes in the Program and Control regions of a 4-Gbyte virtual address space. The user stack is mapped independently into the control region for each process in a system.

All other user code and data (including, for example, "heap" data managed by the NEW and DISPOSE procedures) are mapped in the program region, which is shared by all processes in a job. Data mapped into the program region can be either shared or unshared by a job's processes, depending on how the user chooses to declare them in VAXELN PASCAL.

The kernel does not support demand paging or swapping and, in fact, does not perform I/O to disks or other devices for any reason. Disks and their associated software are needed only when you want them for file storage. The VAX page-fault mechanism is used only for dynamic extension of the user stack. The amount of physical memory on the target computer is the limit for the total size of instantaneously executable VAXELN processes running on a single processor.

Communication and Synchronization Mechanisms Are Key to Concurrency.
Concurrent processes can work successfully only if they manage their access to shared data and physical system resources such as disk storage. Processes communicate using several different VAXELN objects, including ports and semaphores.

Ports are first-in/first-out queues of messages exchanged between processes in a VAXELN system. A port has only one receiver (''reader''), the job in which it was created. However, it can receive messages from any other process in the same VAXELN system or in the same network.

In VAXELN PASCAL, the object is represented by the PORT data type. Its values include both an Ethernet address and object ID thereby uniquely identifying the port to any process at any network location. The values of all other system objects are simple object IDs that are meaningful only in their creating jobs. This means that VAXELN jobs can communicate only by sending messages to one another, not by sharing data or other means. A special port, the job port, is created implicitly when a job is created.

By waiting for a PORT value, a process can synchronize its execution with the arrival of a message at that port.

Names are user-defined names of ports. They ensure the transparency of message-port locations in user programs. A NAME defines a text name and its translation, an associated PORT value. The kernel's only NAME operation, other than creation and deletion, is the TRANSLATE_NAME procedure, which returns the PORT value associated with the text name.

When names are created, they can be specified as local or universal. A local name is meaningful only in its own native VAXELN system (that is, in its creating job and any other jobs running on the same network node). A universal name, however, is meaningful everywhere in the network of VAXELN systems.

The kernel in the caller's system translates the local names. The translation (and maintenance) of universal names is a Network Service operation—one of the reasons why the Network Service is built into every system in a network application.

In summary, the programs in the network can refer to message ports by translating names instead of by using explicit PORT values. Then, if programs are relocated in the network, only the name translations change. Since translations are performed by the kernel or Network Service, the code in the user's programs is unaffected.

Messages are transmissions between two VAXELN processes that can be processes in the same job, in different jobs, in different jobs on different network nodes, and so on. A MESSAGE indirectly specifies the sender's port, the destination port, and the "text" (any data type) of the message.

Events are synchronization objects used to coordinate execution of one or more system processes in response to a specific event.

An EVENT object represents the occurrence of an event. Its state, cleared or signaled, remains the same until explicitly cleared by the CLEAR_EVENT procedure or signaled by the SIGNAL procedure. Other processes can await the event with the WAIT_ANY and WAIT_ALL procedures. When an EVENT object is signaled, all processes waiting for it become ready, if their wait conditions are otherwise satisfied.

VAXELN events are the same, in essence, as event flags in Digital's VAX/VMS and RSX-11M operating systems.

Semaphores are synchronization objects used to execute one process as the result of one signal.

A semaphore acts as a gate controlling access to a resource. It maintains a count of the currently available units of the resource, such as the number of disk drives available, the number of gates available at an airport, or, in the case of an actual railroad semaphore, the "number of tracks" (0 or 1) available to a train going in a particular direction.

You set the initial and maximum counts when you create the semaphore. The semaphore count interacts with the WAIT and SIGNAL procedures to control the execution of processes that wait for it.

Unlike events, semaphores are changed by the action of the WAIT procedures. Specifically, signaling a semaphore increments the count, and, if a process continues as a result of signaling the semaphore, the semaphore's count is decremented. A waiting process does not proceed unless the semaphore count is greater than zero. Together, these operations on the count mean that, at most, one process becomes ready when a semaphore is signaled.

A process that wants to use the controlled resource waits for the semaphore. If the semaphore count is greater than zero, it is decremented, and the process is ready. If the count is zero, the process waits until some other process signals the semaphore.

A process signals a semaphore when it no longer requires the resource; SIGNAL increments the semaphore count. When the count is incremented, the first waiting process is allowed to proceed.

Thus, the maximum count of a semaphore represents the available units of the resource being controlled. For example, if an automated factory has 10 loading docks, truck access to the docks can be controlled with a semaphore whose maximum and initial counts are both 10. The initial count of 10 means that the first 10 trucks will be allowed to proceed immediately to the docks.

One process monitors the use of the docks by trucks and signals the semaphore whenever a truck leaves the dock. (Only one truck proceeds as the result of one signal.) Another process waits for the semaphore before allowing a new truck to proceed toward the dock, perhaps by raising a gate.

With a maximum count of one, a semaphore can be used to guard a single item, such as a shared variable. This type of semaphore, also called a binary semaphore, can be used to prevent more than one process from simultaneously accessing the variable.

The MUTEX data type and procedures for using it are also provided in VAXELN PASCAL. A MUTEX has the same general semantics as a binary semaphore except that a process does not have to call a WAIT procedure to determine whether the MUTEX is currently available. Using MUTEXES, a lock-unlock sequence (for example, getting and relinquishing access to a shared variable) consists of only four machine instructions. This can significantly reduce the time for the operation compared with the use of binary semaphores—typically by a factor of 10. Either binary semaphores or MUTEXES provide completely safe access to shared variables.

Device objects are used to synchronize a device driver program with the completion of an interrupt service routine. Interrupt service routines are written in VAXELN PASCAL. A declaration has the same form as a PASCAL procedure and the word INTERRUPT_SERVICE is used instead of PROCEDURE. They have two parameters: one defines the device registers for the device being driven, and the other defines a communication region in which the interrupt service routine and the driver program exchange all their information. The communication region is simply a dynamic data area that is accessible to both the job and the interrupt service routine because it is mapped in VAX system space.

A DEVICE value relates an interrupt service routine to a description of the device created with the System Builder. The interrupt service routine (ISR) is called by the kernel on the occurrence of a device interrupt and begins execution within two machine instructions of the interrupt's posting. The ISR can take any action needed to service the interrupt, using the device register pointer to gain access to the device registers. In the typical case of a device that interrupts for several reasons, the ISR can examine the control status register to determine which interrupt has occurred.

For instance, the ISR can call SIGNAL_DEVICE to synchronize the main program with the interrupt. An ISR can also handle the special requirements of devices after power failure.

When the interrupt service routine has processed the interrupt, it calls the SIGNAL_DEVICE procedure. The driver program waits for this signal by supplying the DEVICE value to the WAIT_ANY or WAIT_ALL procedure.

All the device drivers supplied with the tool kit are written exclusively in VAXELN PASCAL. As a guide to the user who must write new ones for other devices, sources are provided for each driver.

Using Objects

Each system data type, or object, has a corresponding CREATE procedure, which creates a new object of that type and defines its initial properties. For example, a program creates and initializes an event by the following kind of sequence.

```
PROGRAM a;
VAR
  proceed: EVENT; {EVENT variable}
  .
  .
  BEGIN
  .
  .
  CREATE_EVENT (proceed, EVENT$CLEARED);
    {Create an event value and assign it to proceed. The event is initially cleared.}
  .
  .
END.
```

Here, the EVENT variable proceed holds a new value created by CREATE_EVENT.

Communication

A set of message-passing facilities allows two processes to communicate. Note again that, viewed from above, VAXELN applications are made up of message-passing jobs, and messages are the only means of communication between jobs. The resulting independence of jobs from each other's databases means that they can be redistributed at will among nodes in a network, without changing code.

For example, the File Service, Network Service, disk drivers, and user programs all are independent jobs in an operating VAXELN application. When a user calls the WRITE procedure in PASCAL to write a disk record, messages are transparently exchanged among the I/O runtime code, File Service, disk driver, and, in the network case, the Network Service and kernel. From the program's and programmer's viewpoint, it makes no difference where the disk is actually located in the network.

Messages can be sent directly to a message port, or two ports can be bound together to form a circuit. Circuits have several desirable properties compared with unconnected ports:

- Messages sent over circuits are guaranteed to arrive at the destination in the order in which they were sent.
- Messages can have any length that the sender and the receiver can produce. If the transmission is across the network, the Network Service will divide the message into segments of the proper length, transmit the segments in sequence, and reassemble them at the destination node.
- Flow control prevents a fast sender from overwhelming a slow receiver.
- Circuits can be "opened" as files, allowing PASCAL I/O procedures to be used for message transmission.

No performance penalty arises from using circuits. And because their additional features are needed in most network applications, circuits are the recommended means of sending messages in most applications.

Message transmission between processes on the same network node does not involve moving data. Using VAX memory management, the memory containing the message is unmapped from one process and remapped to another. That means message transmission is fast, predictable, and nearly independent of the message's length.

To send a message, you declare a pointer to the type of data you want to send and supply the pointer to CREATE_MESSAGE. The procedure dynamically allocates a data area of suitable size in the Program region, using the pointer's associated type, and returns its address in the pointer. You then use the pointer to fill in the data area, obtain the ID of the destination port, and, for the case of an unconnected message port, supply the MESSAGE and PORT values to the SEND procedure. For example:

```
VAR mtext: ↑VARYING_STRING(80);
    command: MESSAGE;
    destination: PORT;
.
.
BEGIN
    CREATE_MESSAGE (command,mtext);
    mtext↑ := 'START';
.
.
TRANSLATE_NAME(
    destination,
    'SLAVE_PROCESSOR');
    SEND(command, destination);
    END .
```

The SEND procedure removes the data from your address space and places the message value in the destination port.

The receiver process waits for a message to arrive on its port and then uses the RECEIVE procedure to obtain it. The RECEIVE procedure automatically maps the message data into the receiver's address space and returns a MESSAGE value and a pointer to the message data for the receiver's use.

If circuits are used, the sender establishes the circuit and then sends the message via the circuit:

```
VAR mtext: ↑VARYING_STRING(80);
   command: MESSAGE;
   slave: PORT;
.
.

BEGIN
   CREATE_MESSAGE(command,mtext);
   mtext↑ := 'START';
.
.
CONNECT_CIRCUIT(
   slave,
   DESTINATION_NAME := 'SLAVE_PROCESSOR');
SEND(command,slave);
END .
```

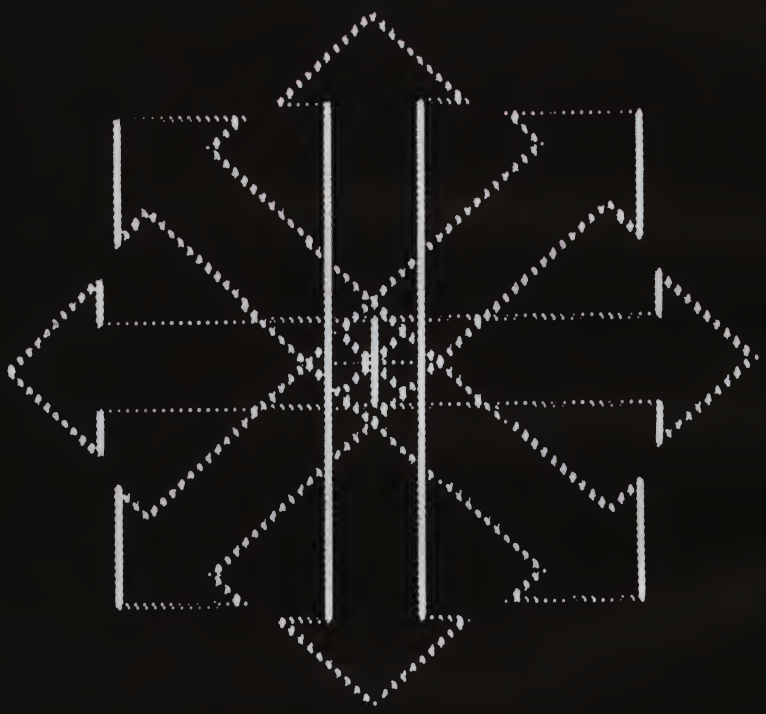
The receiver process, in the circuit case, typically establishes one port—perhaps its job port—to receive connection requests and uses one or more other ports to form the circuits with other requesting processes. For example:

```
VAR mtext: ↑ VARYING_STRING(80);
   command: MESSAGE;
   myport, cport: PORT;
   myname: NAME;
```

```
.
.
BEGIN
   JOB_PORT(myport);
   CREATE_NAME(
      myname,
      'SLAVE_PROCESSOR',
      myport,
      NAME$UNIVERSAL);
.
.
{Wait for circuit requests, then connect cport in the circuit.}
ACCEPT_CIRCUIT(
   myport,
   CONNECT: = cport);
.
.
RECEIVE(command,mtext,cport);
.
.
{Interpret command.}
END .
```

Note that the receiver process must know beforehand the formats of all messages it expects to receive. That is, the sender and receiver must have established a message protocol. For example, if the receiver is a command interpreter of some kind, it must know a set of predefined commands to which it will respond. It can return an error message to the sender (or, more likely, to an operator's terminal) if it receives a message that does not contain a valid command. Defining a protocol is the basic design task in interprocess communication.

Chapter 3· Language: Powerful Programming in PASCAL.



Concurrent PASCAL Replaces MACRO Programming.

The VAXELN PASCAL language is a compatible superset of the language defined in the International Standards Organization document ISO/DIS 7185. Any program written in ISO-standard pascal can be compiled by the VAXELN PASCAL compiler and executed as part of the system.

However, VAXELN PASCAL is really much more. It is supported by a highly optimizing compiler that generates position-independent VAX code. It has been extended to include data types and operations that support both concurrent and system programming. All of these enhancements add up to a high-level language that gives you the advantages of an assembly language such as: widespread access to the hardware, superefficient program execution, and more.

Special Data Types Enhance Programming.
The general design principle for the compiler was to extend the language as little as possible to reflect the VAXELN kernel's data types and operations, to provide a relatively small set of new features that enhance the standard language for system programming, and to completely eliminate the need for assembly language or any other language for VAXELN system programming.

Language Extensions Make VAXELN PASCAL as Powerful as MACRO.
The VAXELN Pascal language is further enhanced through language extensions such as the PROCEDURE_TYPE and FUNCTION _TYPE directives. With them you can declare the parameter types—and for functions, the result type—of entire categories of procedures and functions. Two such types are predeclared:

EXCEPTION_HANDLER and INTERRUPT_SERVICE. These features, in combination with others, such as the separate compilation features, allow you to construct abstract facilities, popular in modern languages like Modula II.

Highlights of the language extensions are:

- Separate compilation allows large programs to be compiled as separate modules, without giving up PASCAL type-checking among modules.
- A generalized mechanism called flexible type, is provided for declaring data with parametric extents. Three such types—STRING, VARYING_STRING, and BYTE_DATA—are predeclared for representing fixed-length character strings, varying-length strings, and series of bytes of uninterpreted data type, respectively. The flexible-type mechanism also is available for user-declared types.
- Procedure and function parameters can be declared with conformant extents. For example, procedures can be easily written to handle arrays of any number of elements determined at compile or run time.
- The user can apply the INLINE attribute to some procedure and function declarations, meaning that a "call" to the routine is expanded inline and does not result in an actual procedure call at the machine-language level.
- Procedures and functions can have optional parameters and argument lists of variable length. In addition, routines can be called with nonpositional argument lists.

- The user can declare procedure types and function types. A function type, for example, represents a class of function that has arguments of the same type and that returns results of the same type. One such type, EXCEPTION_HANDLER, is predeclared; its arguments are the standard VAX exception arguments, and its result type is BOOLEAN, where TRUE means that an exception was handled.

- A number of features are provided for representing and manipulating data exactly and conveniently in system programming problems. Included are typecasting (allowing you to refer to a variable as if it had a specified type), a generalized CONVERT function, attributes to specify the machine representation and alignment of data predeclared procedures for operating on device registers, and the BYTE_DATA type. Also provided is the type ↕ANYTYPE, representing pointers whose associated types can be left unspecified until they actually are used to refer to a data item in storage.

- Interrupt service routines are declared in the language in the form of a PASCAL procedure with the word INTERRUPT_SERVICE in lieu of PROCEDURE. Interrupt service routines can be written in PASCAL (unlike VMS, which requires that they be written in MACRO) and are evoked by the kernel on the occurrence of a device interrupt.

- Process blocks are declared in the language in the form of a PASCAL procedure with the word PROCESS_BLOCK in lieu of PROCEDURE. Process blocks are activated by the kernel procedure CREATE_PROCESS; they are scheduled concurrently with the main program's master process.

- The kernel data types DEVICE, EVENT, MESSAGE, NAME, PORT, PROCESS, and SEMAPHORE are predeclared data types in the language, with full data-type checking in effect. Similarly, kernel procedures are predeclared, with compile-time type checking applied to their arguments.

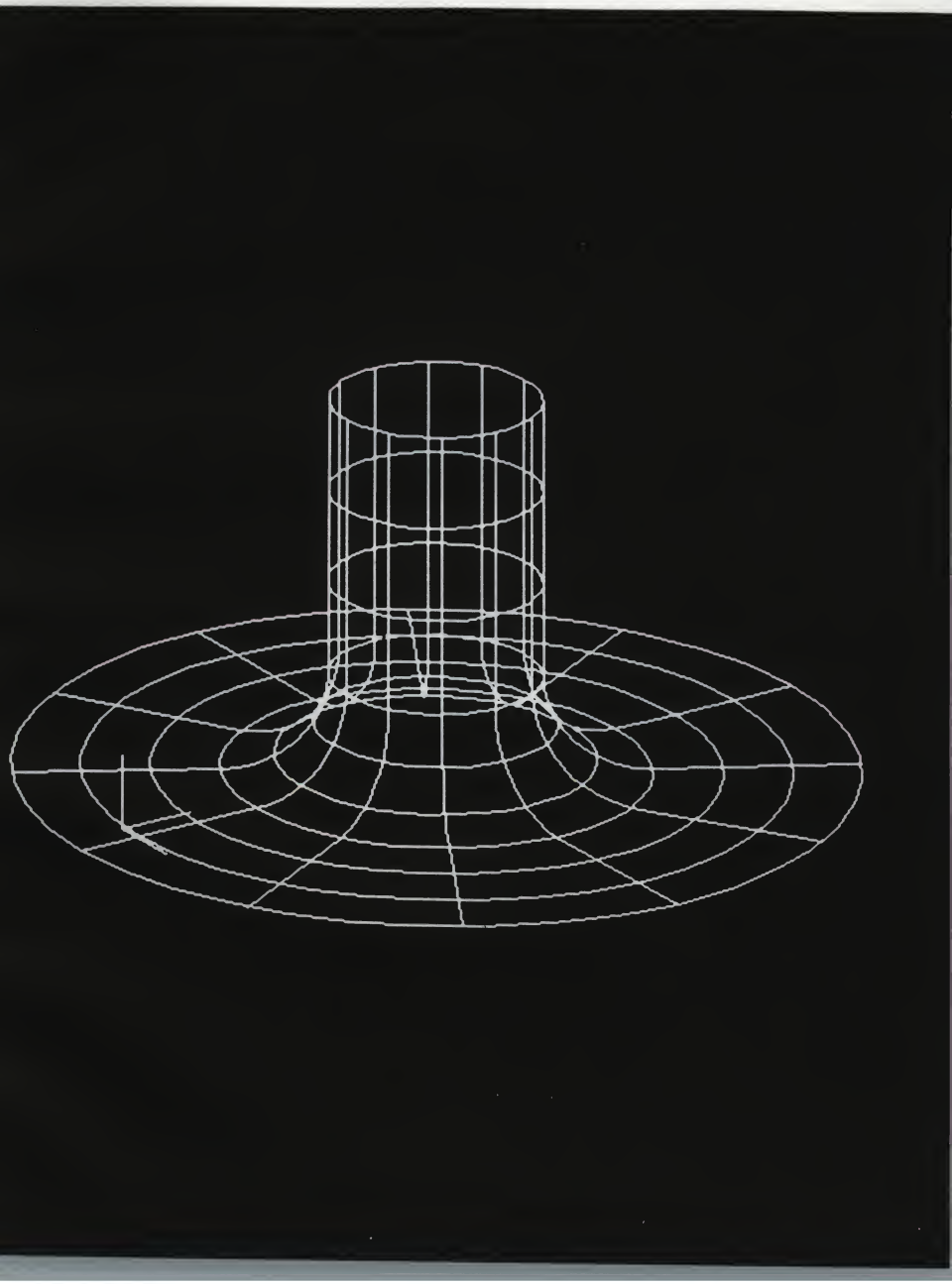
- The MUTEX data type and support routines are provided as library declarations, significantly enhancing the performance of the binary semaphore.
- The QUEUE_ENTRY data type is provided, along with predeclared procedures, for representing and using VAX absolute queues.

A VAX/VMS Compatible File Service.
The VAXELN File Service supports I/O operations from VAXELN PASCAL programs to file-storage devices such as disks, as well as remote file access from DECnet-VAX nodes. I/O requests from the user's PASCAL programs (calls to the I/O runtime routines) are interpreted by the File Service and performed by the appropriate device-driver program. The File Service and the tool kit's disk-driver programs use Digital's Data Access Protocol (DAP), Version 7.0, as their I/O interface protocol. User-written drivers can run compatibly with the File Service by following this protocol, and programming tools are supplied in the tool kit for this purpose.

The File Service uses the same on disk structure as VAX/VMS and the same internal file format as the VAX Record Management Services (RMS). Accordingly, disk volumes from one environment can be read and written on the other. VAX/VMS file utilities can be used from a DECnet-VAX node to manipulate and maintain files on a remote VAXELN system.

File Service files are organized sequentially, but can be accessed either sequentially or randomly. Random access FIND and LOCATE procedures are provided with VAXELN PASCAL.

The File Service is not required for I/O to printers or terminals and it is omitted from VAXELN systems that do not need to operate on local files.



Here's a Complete Tool Kit for Specialized Software Development.
The VAXELN tool kit, which runs on the VMS and MicroVMS operating systems, has everything you need to bring dedicated VAX applications to life as quickly and efficiently as possible.

VAXELN tools include the optimizing PASCAL compiler, the System Builder utility, and the symbolic debugger. VAXELN also includes extensive libraries of prewritten modules that provide a kernel of services for each VAXELN application, as well as prewritten device drivers, file handling routines, and the Network Service for transparent communication across an Ethernet local area network.

VAXELN Applications Include Two Kinds of Programs.
VAXELN applications are divided into two kinds: user programs most often written in VAXELN PASCAL and programs supplied by Digital.

User programs can include user-written device drivers or resource services, as well as the normal computational programs. You never need to write in any language but PASCAL.

Programs supplied by Digital include the File Service and Network Service. Also included in the tool kit are drivers for the commonly needed peripheral devices: disk devices, terminals, lineprinters, Ethernet controllers, and so on. Drivers are provided in source form; you can rewrite, adapt, or modify them to suit your needs.

Developing a VAXELN Application, Step by Step.
You develop a new VAXELN system by writing new VAXELN PASCAL programs or by using existing programs written in standard PASCAL.

If your PASCAL source file specifies a complete program, you can give the resulting object module directly to the VMS Linker utility to produce a program image. Otherwise, the object module can be used as input to later compilations, continuing until you have formed a set of object modules specifying the complete program. Then you can give the entire set to the Linker to prepare the program image.

Then, with the tool kit's System Builder utility, you form an operational system by combining the VAXELN kernel with the programs, using any of the drivers and services you want. The System Builder is a menu-driven program that runs under VAX/VMS. It offers an extremely simple means of describing devices, programs, and overall system characteristics, especially compared with the long "system generation" dialog needed by many other realtime development systems.

It's surprisingly simple and straightforward to create a network of VAXELN systems for distributed processing. You build each system separately, including the VAXELN Network Service in each one, then connect them to the Ethernet by clamping a transceiver onto a cable.

EDT: One of the Best Text Editors There Is.
EDT is an interactive text editor that's easy enough for beginners yet powerful enough for experienced users. Using EDT, the VAX/VMS text editor, you can create your VAXELN source-code files. It lets you do both screen editing and keypad editing on a Digital video terminal.

With EDT you can work on multiple files of text or programs simultaneously. Meanwhile, a journaling facility protects against loss of edits due to system failure.

The EDT HELP facilities are extensive. You get help on general EDT operations by typing HELP or pressing the HELP key in keypad mode. In keypad mode, common EDT functions can be performed on a separate set of keys.

PASCAL Compiler Optimizes Code

The PASCAL compiler does several different types of code optimizations to increase execution efficiency. The result is code that runs as fast as "hand optimized" machine code. Optimizations include removing common subexpressions from statement sequences, placing local variables in multiple registers, removing nonvarying expressions from loops, and replacing certain code patterns with simplified code. All optimizations are optional; you can eliminate specific ones whenever you run the compiler.

Separate Compilations and Flexible Libraries Increase Productivity.

VAXELN provides PASCAL type-checking among separately compiled modules. This means that it's generally not necessary to compile everything needed by a module whenever a new feature is added to the program. As long as the same object modules are used in both compilation and linking, type consistency is assured.

When a module is compiled, items declared at its outer level can be "exported," allowing them to be used in other modules. Exported names, which by default are all the names declared at the outer level, are written in the object module in a special symbol table. When such an object module is included in the compilation of another module, the compiler reads the export symbol table to get declarations as needed. The symbol table has type information for each name, so type checking is retained among separately compiled modules. The module heading also has import and export options that control the use of names between modules.

The VAX/VMS Librarian maintains libraries of modules, including object modules. The VAXELN compiler and VAX/VMS Linker both can use object modules stored in these libraries.

Several libraries of runtime modules are supplied as part of VAXELN. These include, among others, the modules of the Data Access Protocol, the File Service, and the general math routines available in the language.

Modules can be combined into shareable libraries so that, when more than one program image references them, there is only one copy of the code in the system image.

The System Builder Utility Puts Your Application Together.
After creating, compiling, and linking the programs that make up an application, you actually put the executable system together using the VAXELN System Builder utility. The System Builder combines one or more program images into a bootable system image. It also produces a system map, if desired, that lists all the images in the system, including their program descriptions, all devices with their device descriptions, all terminal descriptions, and all the system characteristics.

Menu Driven, Simple Interface for Building a System.
The System Builder includes an EDIT mode that displays a series of menus on VT100-series terminals. It lets you interactively tell the utility what programs, devices, terminal characteristics, and system characteristics you want.

The main menu appears as soon as you enter the System Builder utility. From the main menu, you can build a system from the current data file information; edit the characteristics of the entire system; and add or edit descriptions of programs, devices, and terminal lines.

Configure Your System Quickly and Easily.
With the System Builder, generating a VAXELN application system is just as fast as compiling a program. You select such menu items as *Add Program Description* and *Edit System Characteristics,* to see more specific menus and data screens. Then, modify the appropriate lines to suit your needs; the utility even checks your integer menu entries to see if they're in the allowable range. You finish by returning to the main menu and selecting the *Build System* entry. The System Builder then automatically creates the new system image file and exits.

The Symbolic Debugger Gets Your Programs Running Faster.
There are two ways to debug your VAXELN applications. You can build the onboard debugger into your application and talk to it with the console terminal on the target machine. Or you can run the debugger on your host development system and control your target machine via an Ethernet link.

Both VAXELN debuggers allow you to set breakpoints, evaluate expressions, examine program locations, and deposit values in locations. You can also introduce command definitions and variable declarations that are valid while you are running the debugger.

The debuggers can display lines of source text during the debugging session, and traceback information will show the source-line number where an error occurred. Traceback information shows the recent execution history leading to the error.

When you debug remotely from the host development system, you can refer to names of your PASCAL program variables and routines.

The appropriate debugger is built automatically into your system if you select the debugging option on the System Builder's Edit System Characteristics menu. Selecting this option lets you debug your complete system, including other network nodes.

VAX/VMS Operating System Is Your Host System Software.

The VAX/VMS operating system has become one of the most popular in the superminicomputer industry. It is a powerful software base that organizes the great potential of the VAX hardware architecture. It provides priority scheduling of computer jobs, system management tools, and security and accounting controls. There is a single set of commands for performing timesharing, realtime, batch, and interactive computing.

The VAX/VMS System is highly reliable. Built-in protection mechanisms in both the hardware and software ensure data integrity and system availability. Online diagnostics and error detecting and logging verify system integrity. Many hardware and software features provide rapid diagnosis and automatic recovery, should power, hardware, or software fail.



Transparent Communication Is Clearly Easier.
In a network application, jobs on one system send messages to jobs on other systems across a local area network (LAN). Jobs communicate either through the SEND procedure or through input/output operations that use the services, drivers, and hardware of a different node.

The VAXELN Network Service comes as a program image to be built into each system that participates as a node in the Ethernet network. The System Builder implicitly includes it when you select the remote debugging option for a system under development.

The Network Service Makes Distributed Processing Simple.
The Network Service in each node manages all message transmissions between Ethernet nodes (including those between VAXELN nodes and DECnet-VAX nodes) and maintains a set of universal names among VAXELN nodes. One set of universal names is known throughout the network. Each Network Service uses universal names as unique identification for message sources and destinations.

With the System Builder, the user can specify that the finished system's node is eligible to be the name server for the VAXELN network.

In an operating network, the "nominee" system with the highest Ethernet address is elected the name server. Each system retains the list of universal names it has created and listens for a periodic broadcast from the current name server. If no broadcast is heard during a specific interval, a new name server is elected, and each node sends the new server its list of universal names. This method helps preserve the integrity of universal names in the event of a name-server shutdown.

A VAXELN system is a DECnet end node. It can have one Ethernet datalink controller in its hardware configuration. The VAXELN system can communicate directly over the Ethernet with any DECnet-VAX node on the same Ethernet. If there is a full routing system on the Ethernet, it can communicate through the full-routing system with any other nodes in the entire network.

Since both DECnet and Ethernet have a maximum message size, the maximum size of transmitted VAXELN messages, or datagrams, is specified as a System Builder characteristic. When circuits are used to transmit network messages, as described below, the Network Service segments messages. It divides long messages into datagrams of the proper size and transmits them in sequence.

The Network Service and kernels on the network's nodes transparently communicate with each other to perform the DELETE, CREATE, and TRANSLATE operations on universal NAME objects.

VAXELN Ethernet Operations Are Part of DECnet.
The Network Service uses the Phase IV DECnet Network Service Protocol (NSP) and Session Control Protocol to provide transparent application-level circuits to remote nodes, as described below. The Network Service's NSP module then uses the routing module, also DECnet Phase IV, to deliver messages to remote systems.

VAXELN DECnet nodes can be managed from VAX/VMS with the DECnet-VAX Network Control Program (NCP). The VAXELN Network Management Listener (NML) provides a subset of DECnet Network Management to monitor the network and to control DECnet systems.

The Loopback Mirror can test the Network Service on a VAXELN node from a VMS node or from another VAXELN node. It passively loops messages sent to it using the NCP LOOP NODE command. The Loopback Mirror is a good test of the Network Service's ability to communicate with the rest of the network.

Network Communication Is Transparent
Transparent communication allows VAXELN and VMS programs to exchange information using standard I/O statements, ignoring the fact that they are separated on the network.

When one job sends a message to a job on another node, the Network Service locates the destination in the network and formats the message for transmission across the Ethernet. Neither the sending nor the receiving job communicates directly with the Network Service. Instead, the kernel on each node may determine that, for example, an outgoing message is not destined for any of the message ports on its node and forwards the message to its Network Service. Thus, when jobs on the same system exchange messages, the Network Service is not involved. In non-network applications, it can be omitted from the software system.

Circuits Ensure Reliability.
Binding two message ports together in a circuit provides the most reliable communication. The CONNECT_CIRCUIT procedure connects a sending message port to a specified destination port. The destination port accepts the request to link with ACCEPT _CIRCUIT.

Circuits control the flow of messages between processes in the system, so you don't have to program for it. Once a circuit has been established, the sending process automatically waits if the partner port contains its limit of messages. Using this technique, no sending process can overwhelm a destination port by sending

messages faster than the receiving process can accept them. This protects against unreceived messages "piling up" and consuming excessive memory in the system.

Transparent Networking Makes Distributed Processing Painless.
VAXELN applications can be run on stand-alone VAX and MicroVAX systems or, using networking software provided in the tool kit, connected in an Ethernet local area network (LAN). The LAN can include VAX/VMS nodes or any other nodes using DECnet-VAX services and protocols. VAXELN PASCAL programs can be redistributed unchanged among the nodes in a network.

What is transparent networking? Suppose a network is made up of the nodes FRED, THOMAS, and MARION. A program on node FRED wants to talk to one on node MARION. In a nontransparent network, the program on FRED would have to know the network location of the receiver and, possibly, the network path needed to reach node FRED—for example, "I want to talk to THOMAS - MARION - receiver." An improvement is to have network routing done by the system, although you still must know the receiver's node—"I want to talk to MARION - receiver." In a completely transparent network, such as a VAXELN network, the statement is simply, "I want to talk to receiver." That is, receiver can be identified by a universal name that, from the program's viewpoint, is independent of the program's actual location.

Transparent networking makes VAXELN one of the simplest and most effective ways to implement multisystem applications. It makes server networks very straightforward. In a server network, some or all of the participant systems perform specialized functions for the entire network. For example, all disk devices and data

files might be concentrated at one node. Transparent communication helps the whole network function smoothly with a minimum of programming effort.

Ethernet Is the VAXELN Communication Medium. Ethernet is easy, fast, and economical. It is the least expensive and most effective communication alternative for small geographical areas. It uses baseband signaling, as opposed to broadband signaling. Ethernet nodes transmit a large amount of information across a single, shared network channel using CSMA/CD access control.

Transparent networking makes VAXELN a strong choice for distributed processing applications such as server networks. With networking software provided in the tool kit, VAXELN systems can be connected in an Ethernet local area network, which may include VAX/VMS nodes or any other nodes using DECnet-VAX services and protocols. You add each system to the network simply and directly; there are no complex configuration problems. Communication between systems is literally as simple as communication between programs on the same system.

Chapter 6· VAX Hardware: Systems of Choice.



The VAX Architecture Is a Proven Winner.
VAX computers are high-performance multiprogramming computer systems ideally suited for a wide variety of applications, including realtime, batch, timesharing, and program development. The systems combine 32-bit architecture, efficient memory management, and a virtual memory operating system that provides essentially unlimited program address space.

VAX processors provide 32-bit addressing, sixteen 32-bit general registers, and 32 interrupt priority levels. All except the VAX-11/730 also include an efficient memory cache that increases program execution speed by reducing the need to access memory.

The instruction set includes floating-point, packed-decimal arithmetic, and character-string instructions. Many of the instructions are direct counterparts for high-level language statements. The software system supports programming languages that take advantage of these instructions to produce extremely efficient code.

The VAX/VMS virtual memory operating system provides a multiuser, multilanguage programming environment on the VAX hardware. The floating-point instructions, efficient scheduler, and optional VAX-11 FORTRAN language (one of many available VAX/VMS languages) are ideal for realtime and scientific computational environments. The information management services, large-capacity peripherals, and another optional language—VAX-11 COBOL—make the system equally suited to commercial applications.

To ease the sometimes complex task of configuring a computer system, Digital has developed the System Building Block (SBB) concept. Using it, you start with a basic system kernel, then choose other components from a menu. It offers an extremely

simple means of describing devices, programs, and overall system characteristics, especially compared with the long and complicated "system generation" dialogs needed by many other development systems. If you prefer to configure your system without using the SBBs, you can.

VAX Systems Have Everything in Common.
VAX architecture ensures software compatibility across VAX systems through common attributes such as addressing modes, the instruction set, and data types.

The processor executes variable-length instructions in native mode and nonprivileged PDP-11 instructions in compatibility mode. Native mode includes the PDP-11 data types. It uses addressing modes and instructions similar to their PDP-11 counterparts, and the software supports compatible languages, and file and record formats.

PDP-11 users will find the native VAX-11 instruction set and programming characteristics easy to learn when developing new applications for the VAX system.

MicroVAX I: Puts VAXELN to Work on Micro Applications.
The MicroVAX I computer, the first VAX system on a Q-bus, allows VAX users to realize the savings inherent in Q-bus options and packaging. Designed for realtime and timesharing applications, the MicroVAX I system is a low-cost, high-performance, 32-bit processor that is available in several configurations to allow customized system design for specific applications.

The MicroVAX I computer is in the same performance class as the VAX-11/730 and the PDP-11/44 systems. It is available with either 256- or 512-Kbyte parity memory.

One of the challenges in producing the MicroVAX I product was to reduce hardware costs yet maintain VAX-level performance. Part of the solution was to emulate some little-used instructions and data types using microcode-assisted software. The CPU hardware is especially designed to make this emulation so efficient that MicroVAX I can give you VAX performance at a size and price never before seen in a VAX system.

Specifically, decimal and packed-decimal math instructions were moved to software, as well as some string instructions. You choose a hardware double-precision floating point (D or G). The other is emulated, as is the 128-bit format (H).

The MicroVAX architecture specifies that physical addresses can be up to 30 bits long. A physical address on MicroVAX I is 23 bits, allowing a physical address space of 8 Mbytes.

MicroVAX is also fully software-compatible with the existing line of VAX systems, when executing native-mode programs. Non-privileged native programs that run on a VAX processor today will run without change on the MicroVAX I.

VAX-11/725: the Smallest VAX on the UNIBUS.
The VAX-11/25 system is Digital's smallest UNIBUS VAX/VMS system. It is a VAX-11/730 computer packaged in an open-office cabinet along with the RC25 disk subsystem. Compact, quiet, and transportable, the VAX-11/725 system supports up to eight users yet is only 40 percent of the physical size of the VAX-11/730. This makes the VAX-11/725 computer appealing as either a single-user workstation, an inexpensive multiuser system, or a network concentrator.

It is available in three configurations with up to 3 Mbytes of memory. Options include a floating-point accelerator that gives users a faster, more efficient means of processing numerical data.

The RC25 disk subsystem, with a total of 52 Mbytes of storage, provides the reliability and performance of Winchester technology with the flexibility of removable media. Of the 52 Mbytes, the first 26 are on a fixed platter, while the remaining 26 are stored on an 8-inch removable cartridge. This allows the copying of disks at a one-to-one ratio. The fast backup also saves time and media costs. The RC25's intelligent controller adds to the impressive performance of the VAX-11/725 processor by offloading disk handling from the CPU.

VAX-11/730: When Expandability Is the Key.
An ideal distributed processing network node, department-level host, stand-alone computer, front-end machine, remote concentrator, or dedicated program development system, the VAX-11/730 computer is two-thirds the size and about half the price of our next larger processor, the VAX-11/750. Our most flexibly configured VAX, the VAX-11/730 system supports up to 24 users.

Bit-slice and Programmed Array Logic (PAL) technologies and advanced packaging combine to achieve dramatic economies of price and space. Choose a low-cost, rack-mountable chassis or a prepackaged VAX-11/730 system. Or specify a VAX-11/730 system from System Building Block (SBB) menu selections. Both the rack-mountable system and the packaged system include 1 Mbytes of memory (expandable to 5 Mbytes) and a 12-slot backplane. The SBB cabinet and menu systems include 2 Mbytes of main memory (expandable to 3 Mbytes) and nine expansion slots. In both cases, there are plenty of backplane slots for you to add the options of your choice, such as a floating-point accelerator that increases performance in engineering, scientific, laboratory, and other numeric intensive environments. The VAX-11/730 supports a wide range of Digital-supported and customer-developed peripherals and communications devices. Your choice of system configuration and memory options determines expansion capability and price.

VAX-11/750: For More Demanding Work.
The VAX-11/750 computer is the midrange member of the VAX family. Total software compatibility with other VAX computers, impressive performance, and competitive price make it ideal for small firms and for individual departments within large organizations—anywhere that large-system capability is needed by users with restricted budgets or workspace. The VAX-11/750 computer supports up to 64 users in environments such as banking, education, and engineering, and it is the most economical way to start a VAXcluster configuration.

Fewer components account for the reliability and low-cost of VAX-11/750 hardware. Ninety percent of the VAX-11/750 hardware logic is implemented in custom, low-power gate-array circuits. A single gate-array chip performs the functions of 25 standard integrated circuits.

The CPU provides sixteen 32-bit user-programmable registers that can serve as temporary storage, accumulators, index registers, and base registers for user application programs. A virtual address space of 4 Gbytes allows large programs to be run without overlays. Depending upon the configuration, VAX-11/750 systems can include up to 8 Mbytes of MOS memory.

The console subsystem consists of a TU58 cartridge tape drive and a DECwriter terminal.

Options include a floating-point accelerator, extended-range floating-point microcode, additional memory modules of 1 Mbyte each (up to a total of 8 Mbytes), and memory battery backup.

The processor has a 4-Kbyte, direct-mapped, write-through cache that greatly reduces effective CPU access time. The write-through feature protects the integrity of memory because memory contents are updated immediately after the processor performs a write.

The optional User Control Store includes 10 Kbytes (1 Kbyte of 80-bit microwords) of writable storage that allows customers to augment the speed and power of the basic machine with customized microcode functions.

VAX-11/780: Power and Capacity for the Biggest Jobs. The VAX-11/780 computer is the machine of legend, the standard by which all 32-bit machines are measured. The parent product of the entire family, it supports more than 100 users. As a stand-alone system or as part of a VAXcluster, the VAX-11/780 gives you the high performance you need for data center processing. It can also be linked to other VAX systems in local-area or wide-area networks so that data can be accessed and shared by users throughout the organization.

The basic system consists of the VAX-11/780 CPU, 2 Mbytes of memory, a UNIBUS expansion cabinet, and the VMS operating system. Options include a floating-point accelerator, extended-range floating-point microcode, a writable control store and tools, up to 32 Mbytes of memory, and memory battery backup.

Proven Schottky TTL technology gives you the performance and overall system power that you need for your most demanding applications.

Memory is expandable to 8 Mbytes of ECC MOS memory (plus 4 Mbytes of Multiport Memory). The MA780 Multiport Memory option can be shared by up to four VAX-11/780 systems for high throughput.

The 8-Kbyte cache memory, a key to the processor's superior performance, delivers a 95-percent hit rate. An 8-byte instruction prefetch buffer further improves processor performance by prefetching and decoding data in the instruction stream. The control logic fetches longword data from memory to keep the 8-byte buffer full.

The processor's translation buffer contains 128 virtual-to-physical page address translations. This significantly reduces the time spent on such translations.

In addition to sharing the identical set of system software with the VAX-11/750 system, the VAX-11/780 computer also offers compatibility with the peripheral set via the UNIBUS and optional MASSBUS adapters. Up to four UNIBUS adapters can connect Digital- and user-designed peripherals, and a maximum of four MASSBUS adapters can be used for connection of high-throughput disk and tape devices.

VAX-11/782 Attached Processor: Superior Performance for Multistream Applications.
The VAX-11/782 system has the power to run many large computational jobs concurrently and still provide good response to terminal users. Technical, commercial, and OEM (original equipment manufacturer) users can, for example, run their largest programs on the system concurrently with interactive users. Additional applications include financial modeling, experimental data reduction, file transfer, electronic and mechanical design using interactive graphics, and business forecasting.

Consisting of two VAX-11/780 CPUs linked through MA780 shared memory, the VAX-11/782 system supports up to 8 Mbytes of shared memory. It is available as either a complete packaged system or as an upgrade for a single VAX-11/780 computer. In many applications the VAX-11/782 system doubles the performance of a single VAX-11/780 computer—at a small incremental investment.

All peripheral devices connect to the primary processor. The attached CPU provides additional computational resources, and applications can be transported to the attached processor without change. The assignment of tasks to a processor by the VAX/VMS scheduler is completely transparent to the user, so programmers can concentrate on their programs, not on where they're being run.

VAX-11/785: The Top of the VAX Performance Line.
The VAX-11/785 computer, the newest member of our VAX-11/780 series, is also our most powerful single processor—giving you the fastest performance yet from the industry-standard VAX family. With added performance and variety, VAX-11/785 configurations can serve more than 100 users and act as the host for a network. A large 32-Kbyte cache memory benefits users running applications such as CAD/CAM and simulation, wherein programs contain many complex subroutines.

The VAX-11/785 system increases the performance of the VAX-11/780 computer through advances in circuitry design. These advances speed data flow through the processor, improving response time for realtime applications and increasing the capacity for computation in compute-bound applications.

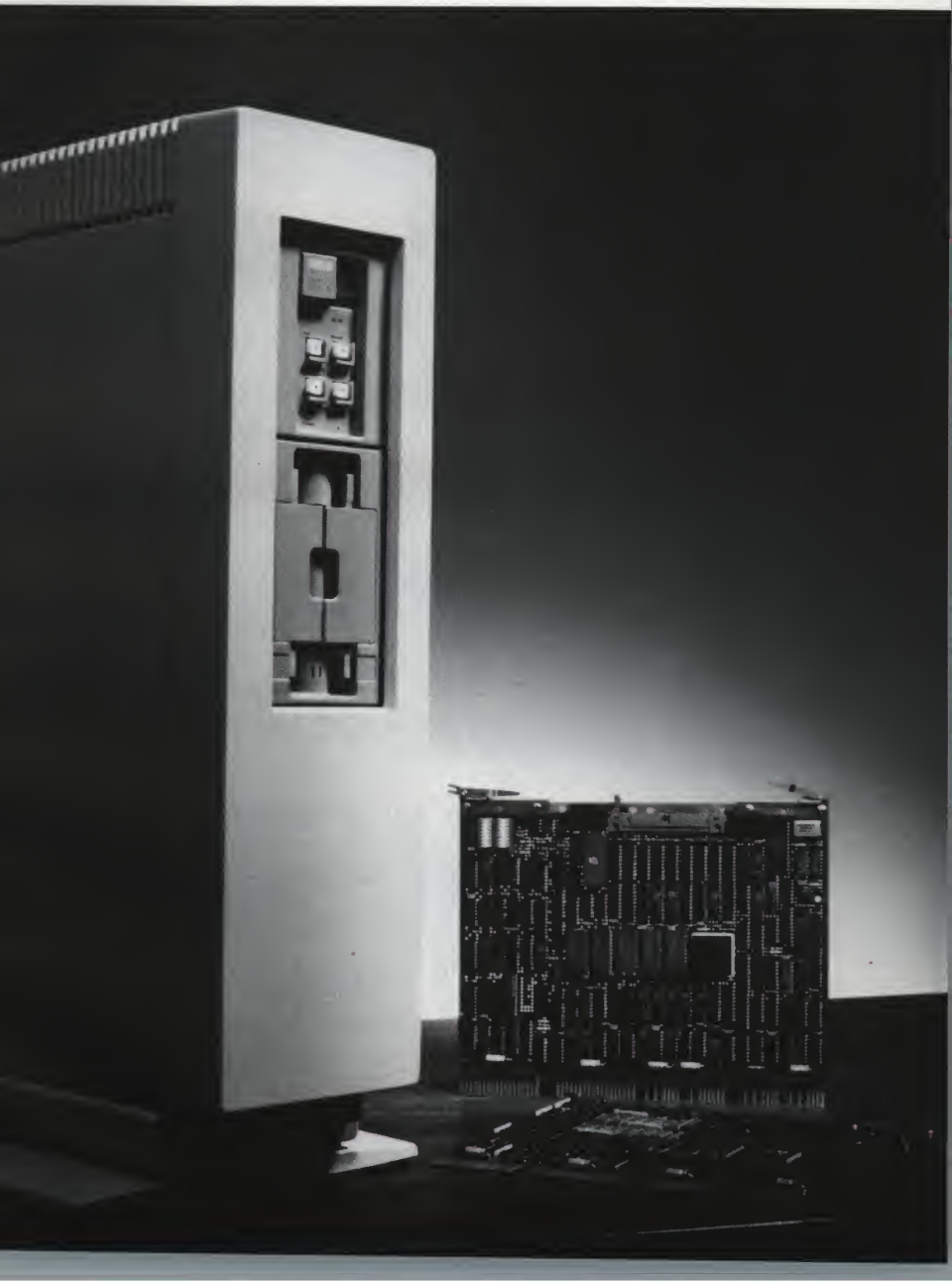
Floating-point G and H data types are standard on the VAX-11/785 system and, as an option, you can add a floating-point accelerator for greater performance in numeric-intensive environments.

VAXclusters: An Industry Original.
A VAXcluster configuration is a unique multiprocessing system consisting of several VAX computers. Up to 16 VAX-11/750 or VAX-11/780 series processors and intelligent storage subsystems function as a single powerful system. Supported by our versatile VMS operating system, cost-effective VAXclusters offer resource sharing and an efficiency that is unmatched in the industry. Users can access any data in the VAXcluster, whether it is stored on local disks or on the HSC50 intelligent mass-storage subsystem.

With several processors sharing the load, you gain speed, protect your data, and optimize disk storage. System advantages include redundancy to maintain data availability if a fault occurs; global data sharing for easy data updates and access, high performance, and multiple data paths that protect against downtime.

Choose from VAXcluster System Building Blocks to build a system to your exact specifications. VAXcluster systems permit expansion of the configuration, as needed, and without interruption of the existing systems or programs. Adding processing power or storage with a VAXcluster will expand your present system, preserving your investment. Or, if you are planning to buy your first VAX, you can start with one VAX-11/750 or VAX-11/780 series processor and add VAXcluster capabilities as your organization grows. You'll be building a VAXcluster system that offers more protection for your data than has ever been available before.

Chapter 7 · Service and Support



Support That Equals Product Excellence.
As you might expect, Digital's Customer Services Organization is one of the largest and most effective in the computer industry. With hundreds of locations around the world and thousands of skilled professionals, Digital offers a wide range of services.

Hardware and software specialists assist customers in analyzing and designing systems, in modifying Digital's software to meet special needs, and in developing customer application software. They can create specific system designs using our standard products or suggest alternate solutions for special situations.

If a customer's needs are not met by Digital's standard offerings, Computer Special Systems analysts, engineers, programmers, and manufacturing specialists can be called in to help provide the necessary hardware and software.

For your day-to-day operations, Digital maintains a complete line of supplies. Everything from magnetic media to word processing supplies can be ordered through our DECdirect catalog, over the phone or by mail.

For your other support needs, Digital offers a choice of software, hardware, and educational service programs.

Service Contracts Tailored to Your Software Needs.
To ensure continued software support and maintenance, Digital provides comprehensive system support services through two contractual agreements—Basic and DECsupport. These new programs are available in the United States and will soon be available worldwide. Check with your Digital representative to determine availability in specific areas outside the United States.

Basic Agreement features include telephone support, software product and documentation updates, access to the Digital Software Information Network described below, and periodic newsletters.

DECsupport is designed for customers who require uninterrupted system performance. It incorporates all services included in the Basic Service, plus the services of a software specialist, remedial support and scheduled preventive maintenance are also included for critical situations.

Digital's *Software Information Network,* part of the DECsupport package, keeps your staff informed of the most recent software information. When you dial into this online network, you have access to extensive databases containing the latest information regarding software solutions, answers to common software questions, and examples of common usage situations. In addition, you can submit your software questions, concerns or solutions via the network so you and other customers can benefit from Digital's expertise.

With two clearly defined packages, you can suit your individual needs and budget. You will know, in advance, what your costs will be. This means you can budget accurately and be assured that you will get the level of software service and training you need for all Digital software.

For most software configurations, the package cost will be significantly lower than purchasing the services and training separately.

Hardware Services That You Can Design.
Digital's hardware support services can also be sculpted to suit the needs of each customer. Two onsite agreements, Basic and DECservice, are available to VAX customers.

Basic service provides next day response and continuous effort of repair during coverage hours. Other features of this package include preventive maintenance, parts, labor, materials, installation of engineering modifications, assigned service representatives, and a comprehensive site-management guide. Basic service also includes problem escalation, which establishes time frames for each level of service personnel attempting repairs. If the problem is not resolved within the allotted time, it must be referred to the next highest level for resolution. The process continues until the problem is solved.

DECservice is a comprehensive onsite service package designed for the customer who requires up to 24-hour, 7-day-a-week coverage. Part of the agreement is a written commitment to respond to your call for service within a specified time. Scheduled preventive maintenance, parts, labor and materials are also provided.

For VAX-11/785, VAX-11/782, VAX-11/780, and VAX-11/750 system users, Remote Diagnosis (RD) is a standard support feature. Customer Runnable Diagnostics (CRDs) is another service innovation available for VAX-11/730 systems.

For customers who maintain their own computers, Digital services include spares-inventory planning; maintenance documentation service; spares, tools, and test equipment. Digital has also assembled well-defined spares kits and a recommended spares list that help customers establish proper spare-parts inventories.

Education Services Show You How.
At 26 Training Centers around the world, Digital offers customers comprehensive educational programs to train personnel before, during, and after installation.

Digest, a quarterly publication of Educational Services, contains a complete schedule of all the software courses offered at the North American Training centers. If you would like to receive a Digital *DIGEST*, contact your nearest Educational Services Training Center or ask your sales representative.

For More Information . . .
If you have more questions about VAXELN, contact your sales representative. For more information on related products, ask your Digital sales representative. Or if you prefer, write to:

Media Response Manager
Digital Equipment Corporation
200 Baker Avenue CFO1-2/M94
Concord, Massachusetts 01742

Appendix I

VAXELN, Ethernet, and VAX Rdb/ELN Are More Than a Match.

The combination of VAX Rdb/ELN, VAXELN and Ethernet provides a unique tool for establishing and maintaining a relational database. VAX Rdb/ELN, which does not require a general operating system, is a small, high-performance relational database management system that runs on VAXELN target systems and provides an alternative to the VAXELN sequential file system. It is part of the VAX Rdb product set, which also includes VAX Rdb/VMS.

Using Rdb/ELN and VAXELN on an Ethernet local area network (LAN) you can develop a database that is shared by multiple nodes.

Without sacrificing CPU time or storage, or adding overhead, you can give every node on a network access to your database. The VAX Rdb/ELN database also can accept data from different sources even if the data is organized in different ways.

Because of the unique relationship between VAX Rdb/VMS and VAX Rdb/ELN, described below, you could even use databases on MicroVAX systems running VAX Rdb/ELN to augment databases located on larger VAX computers.

Rdb/ELN for Time-Critical Applications.

You can develop narrowly focused VAXELN realtime applications that take advantage of the small kernel executive, the VAXELN time-critical response, and the Rdb/ELN database management features.

For example, a satellite passing over an earth station every 50 minutes bursts large amounts of data in a short time to dedicated data collection computers. These processors (possibly PDP-11 and LSI-11) store the data in sequential files.

A MicroVAX, running Rdb/ELN, can be triggered by control information provided by incoming data. For example, the data collection devices signal VAXELN after every 20,000 records that it is time to invoke Rdb/ELN. Rdb/ELN then starts up a data storage process, loading the records from the sequential files into indexed relations for subsequent analysis.

In this example, the programmer who designed the realtime collector system didn't have to be concerned about the database interface because it is already included in VAX Rdb/ELN. Because all the database accesses are handled by VAX Rdb/ELN on the MicroVAX, the programmer also didn't have to worry about the collection process being interrupted by other systems trying to access the data during capture and thus losing incoming information.

A Closer Look at VAX Rdb.

VAX Rdb is a family of relational database management systems that support the Digital Standard Relational Interface (DSRI).

Because of the DSRI, programs written for a VAX Rdb/VMS database are also VAX Rdb/ELN database programs. This means an application running on a MicroVAX under VAXELN can access and use data from a VAX Rdb/VMS database residing on any other VAX. And applications designed to use data in a VAX Rdb/VMS database also can access data from a VAX Rdb/ELN database. All current and future Digital relational products conform to this architecture.

With their remote database management facilities, VAX Rdb products can be configured with any combination of different-sized VAX computers to give you more options in designing your own solutions. Remote management features also mean that the user and the application program do not have to be located with the data; they can be anywhere on the network.

Like all VAXELN applications, Rdb/ELN programs and databases are developed on a host VMS system. When developing a new system, or when no VAXELN system is available, programmers can build a database on a VMS system for program development. When additional programs are added to an existing application, the development system can remotely access a database on a VAXELN system. The development option, which runs on the VMS operating system, is priced separately from the target system license.

What Makes a Relational Database System?

A database management system is a package that allows controlled sharing of data, and helps protect data from corruption caused by system and program errors. Several users can update a database simultaneously without over-writing each other's changes or seeing changes that might be temporary. Within a program, statements can be grouped into transactions. The database system insures that changes made in a transaction are all made or all undone cleanly.

A relational database management system uses the relational datamodel. Data is stored in simple records, without repeating groups or variants. Records are grouped, by type, into relations. A relational database system includes normal storage and update operations (STORE, ERASE, MODIFY). The major retrieval operation in a DSRI database is a FOR loop, which iterates over a group of records called a record stream. The records in the stream can be restricted with a SELECTION clause, which specifies values for fields, etc. A record stream can be made up of records from several relations, put together with a JOIN clause. Duplicate values can be eliminated from a stream with a PROJECT clause.

One major advantage of DSRI databases is that a program running on one node can access databases anywhere on the network. The remote interface is designed to reduce the network load for distributed applications.

Rather than passing relations or the whole database to the host program, the remote interface passes the request from the host to the database management system on the remote node, and returns the results. DSRI and the common remote interface allow a single application to access databases running under VMS, MicroVMS, or VAXELN software.

And because VAX Rdb software lets you transport applications from one VAX computer to another without modification, you gain flexible database management, flexible design, and the ability to distribute and balance application workloads in the most effective way.

Some Other Vax Rdb Features.

1. VAX Rdb can store unstructured data. Text, images, graphics, digitized voice, seismic data, any string of bytes of any size can be stored in a segmented string field in a database. Structured information identifying the string (time, date, read-out station, etc.) can be associated with the segmented string.

2. Rdb/ELN includes a data definition compiler that creates databases and makes changes to database structures. If you are changing the database structure there is no need to make the database unavailable to other users or to unload and reload data.

3. Rdb/ELN includes an interactive query language that is designed for low-volume data access and update, testing data manipulation algorithms, and learning to use the system. It includes online help, automatic command completion, and, when used remotely from the development VMS system, a command and data editor.

4. Data manipulation statements are included in an ordinary EPASCAL program, freely mixed with native language statements. The VAX Rdb/ELN EPASCAL PASCAL preprocessor recognizes the database extensions, translates them into database calls, and provides all the structures and buffers required.

5. Unlike traditional database management systems, VAX Rdb can perform queries and support changing data relationships as requests are made by users and applications.

6. Using the integrated data dictionary, the database can store and maintain definitions as they are created. The dictionary also makes it easy to restructure the database, and allows remote access requests to get at data definitions. This is especially useful, for example, when using VAX DATATRIEVE to access a remote VAX Rdb/ELN database.

7. Validation rules for fields can be included in Rdb/ELN database definitions. When data is stored or modified, the database automatically verifies that the new data is valid.

8. Rdb/ELN fields can also have a MISSING value delared. When a record is stored without specifying a value for some field, the declared missing value is used. The database system eliminates missing values from statistical computations like TOTAL, AVERAGE, and COUNT.

9. The data manipulation language, EPILOG, includes the relational database operators described earlier (STORE, MODIFY, ERASE, JOIN, PROJECT) and a very complete set of comparison operators. In addition to the traditional =, <, >, >=, <>, EPILOG includes BETWEEN and string comparitors CONTAINING, MATCHING, and STARTING WITH. It also includes

arithmetic operators for addition, subtraction. multiplication, and division; a string concatenation operator; boolean operators; statistical expressions; and existence test; and a test for uniqueness.

All these operations are performed in the database, reducing the amount of program code. Programs are easier to maintain and smaller, a real advantage in the VAXELN environment.

Appendix II

Table of Kernel Procedures.

Call Format - Meaning

ACCEPT_CIRCUIT (myport,
FULL_ERROR: = boolean,
CONNECT: = connect_port,
ACCEPT_DATA: = varying_string,
CONNECT_DATA: = varying_string,
STATUS: = integer_variable)

Establish circuit between myport and originator of connection request; if the full error is disabled (FALSE, default), SEND will wait implicitly when the partner port is full (otherwise, an error status is returned by SEND); the varying strings supply optional data to the originator (accept) or receive data (connect). The optional connect_port specifies a different port on which to make the actual connection.

KER$ALLOCATE_MAP (status, register, number, count device_object, spt_address)

Allocate count UNIBUS map registers, beginning with register number, for device_object, and return pointer to first (ANYTYPE) in register. The optional INTEGER variable status receives the completion status, and spt_address receives an ANYTYPE pointer to the system page table base.

KER$ALLOCATE_PATH (status, register, number, device_object)

Allocate UNIBUS buffered datapath for device_object, returning register number, and pointer to datapath register (ANYTYPE) in register. The optional INTEGER variable status receives the completion status.

ALLOCATE_MEMORY (mempointer,
size, VIRTUAL: = address,
PHYSICAL: = address,
STATUS: = integer_variable)

Allocate size bytes of memory, beginning at virtual or physical address, and return ANYTYPE in mempointer.

CLEAR_EVENT (event,
STATUS: = integer_variable)

Set state of event to
EVENT$CLEARED.

CONNECT_CIRCUIT (myport,
DESTINATION_PORT: = port,
DESTINATION_NAME: = name,
FULL_ERROR: = boolean expression,
CONNECT_DATA: = varying_string,
ACCEPT_DATA: = varying_string,
STATUS: = integer_variable)

Request circuit connection between myport and destination name or port; if the full error is disabled (FALSE, default), SEND will wait implicitly when the partner port is full (otherwise, an error status is returned by SEND); the varying strings supply optional data to the destination (connect) or receive data when the destination accepts. (The destination must be specified, either by NAME or PORT value.)

CREATE_DEVICE (device_name,
device_variable,
VECTOR_NUMBER: = integer,
SERVICE_ROUTINE: = isr,
REGION: = rpointer,
REGISTERS: = regpointer,
ADAPTER_REGISTERS: = adpointer,
VECTOR: = vpointer,
PRIORITY: = integer_variable,
POWERFAIL_ROUTINE: = isr,
STATUS: = integer_variable)

Connect to device_name interrupt and return the DEVICE value in device_variable. The vector number is an integer from 1 (default) to 128. Interrupt service routines (isrs) can be supplied for interrupt and power recovery handling. The variable rpointer receives a pointer to the communication region: regpointer and adpointer receive pointers to the first device control register and first adapter control register, respectively; vpointer receives a pointer to the interrupt vector. Integer variables receive the interrupt priority level of the device and the completion status of the procedure. The device_variable can be a scalar DEVICE variable or an ARRAY [O..n] OF DEVICE, where n<-16.

CREATE_EVENT (event_variable,
initial_state,
STATUS: = integer_variable)

Create an event with initialstate EVENT$SIGNALED or EVENT$CLEARED, and return EVENT value in event_variable.

CREATE_JOB (port_variable,
program_name, argument-list,
NOTIFY: = exit_port,
STATUS: = integer_variable)

Create a job running program_name, with optional arguments supplied to the program. The variable port_variable value receives the value of the new job's job port. The value exit_port supplies a port that receives notification of the job's termination.

CREATE_MESSAGE (message_variable,
data_pointer,
STATUS: = integer_variable)

Create a message with a data area suitable for the associated type of datapointer, and return the MESSAGE value in message_variable.

CREATE_NAME (name_variable,
string, port, TABLE: = scope,
STATUS: = integer_variable)

Make string the name of port, with scope NAME$LOCAL (default) or NAME$UNIVERSAL, and return the NAME value in name_variable.

CREATE_PORT (port_variable,
LIMIT: = integer,
STATUS: = integer_variable)

Create a message port able to hold up to integer (default 4) messages, and return the PORT value in port_variable.

CREATE_PROCESS (process_variable,
process_block_name,
argument-list,
EXIT: = integer_variable,
STATUS: = integer_variable)

Create a subprocess running process_block_name, with optional arguments supplied to the subprocess, and return the PROCESS value in process_variable. Optional integer variables receive the final (exit) status of the subprocess and the procedure's completion status.

CREATE_SEMAPHORE (sem_variable,
initial_count,
maximum_count,
STATUS: = integer_variable)

Create a semaphore with the specified initial_count and maximum_count, and return the SEMAPHORE value in sem_variable.

CURRENT_PROCESS (process_variable,
STATUS: = integer_variable)

Return the value of the current process in process_variable.

DELETE (system_value,
STATUS: = integer_variable)

Delete the DEVICE, EVENT, MESSAGE, NAME, PORT, PROCESS, or SEMAPHORE value from the system.

DISABLE_ASYNCH_EXCEPTION
(STATUS: = integer_variable)

Disable delivery of asynchronous exceptions to the current process.

DISABLE_SWITCH
(STATUS: = integer_variable)

Disable process rescheduling in the current job.

DISCONNECT_CIRCUIT (port,
STATUS: = integer_variable)

Break circuit, where port is the one in the current process.

ENABLE_ASYNCH_EXCEPTION
(STATUS: = integer_variable)

Allow delivery of asynchronous exceptions to the current process.

ENABLE_SWITCH
(STATUS: = integer_variable)

Resume process rescheduling in the current job.

EXIT (exit_status: = integer,
STATUS: = integer_variable)

End current process, with optional integer exit status delivered to creator.

KER$FREE_MAP (status, count,
number, device_object)

Free count UNIBUS map registers previously allocated by ALLOCATE_MAP. The optional status is an INTEGER variable.

KER$FREE_PATH (status, number,
device_object)

Free UNIBUS datapath number, previously allocated for device_object by ALLOCATE_PATH. The optional status is an INTEGER variable.

FREE_MEMORY (size, virtual_address,
STATUS: = integer_variable)

Free size bytes of memory at virtual_address, previously allocated by ALLOCATE_MEMORY.

GET_TIME (large_integer_variable,
STATUS: = integer_variable)

Return current time in large_integer_variable.

INITIALIZATION_DONE
(STATUS: = integer_variable)

Inform kernel that current process has completed initialization sequence.

JOB_PORT (port_variable,
STATUS: = integer_variable)

Return PORT value of current job port in port_variable.

RAISE_EXCEPTION (argument-list,
STATUS: = integer_variable)

Raise exceptions in current process.

RECEIVE (message_variable,
pointer_variable, port,
SIZE: = integer_variable,
DESTINATION: = port_variable,
REPLY: = port_variable,
STATUS: = integer_variable)

Receive message from port, return its MESSAGE value in message_variable and a pointer to its data part in pointer_variable; integer variables optionally receive the data area's size in bytes and the procedure's completion status; port variables optionally receive the destination port specified by the sender and a port for replies.

RESUME (process,
STATUS: = integer_variable)

Resume rescheduling of previously suspended process.

SEND (message,port,
SIZE: = size,
REPLY: = port,
STATUS: = integer_variable)

Send message to port, optionally specifying the data area's size in bytes and a port for replies.

SET_JOB_PRIORITY (integer,
STATUS: = integer_variable)

Set priority of current job to integer (0_31, 0 highest).

SET_PROCESS_PRIORITY (process,
integer, STATUS: = integer_variable)

Set priority of process to integer (0_15, 0 highest).

SET_TIME (nonnegative_large_integer,
STATUS: = integer_variable)

Set current time to nonnegative_large_integer.

SIGNAL (value,
STATUS: = integer_variable)

Signal value of type EVENT, SEMAPHORE, or PROCESS.

SIGNAL_DEVICE
(DEVICE_NUMBER: = integer,
STATUS: = integer_variable)

Signal DEVICE value from interrupt service routine; the integer optionally supplies the index of the value to signal in an ARRAY [1..n] OF DEVICE, where n<-16.

SUSPEND (process,
STATUS: = integer_variable)

Suspend rescheduling of process.

TRANSLATE_NAME (port_variable,
string, scope,
STATUS: = integer_variable)

Translate string, searching in scope NAME$LOCAL, NAME$UNIVERSAL, or NAME$BOTH, and return the associated PORT value in port_variable.

WAIT_ALL (system_value-list,
RESULT: = integer_variable,
TIME: = large_integer,
STATUS: = integer_variable)

Make calling process wait for all DEVICE, EVENT, PORT, PROCESS, or SEMAPHORE values in system_value-list to satisfy the wait. Zero to four system values can be specified; the optional integer_variable receives 1 if the system values satisfied the wait or 0 if the procedure timed out. The optional large_integer specifies a time interval or absolute time defining the timeout; the timeout is irrelevant, and the wait result is 1, if the necessary conditions were satisfied before the call.

WAIT_ANY (system_value-list,
RESULT: = integer_variable,
TIME: = large_integer,
STATUS: = integer_variable)

Make calling process wait for any value in system_value-list to satisfy the wait. If more than one system value is specified, the optional integer_variable receives the position of the argument satisfying the wait or 0 if the procedure timed out. The optional large_integer specifies a timeout as for WAIT_ALL.

Appendix III

The Lineprinter Driver.

The program LCDRIVER supplied with the tool kit shows several features of VAXELN PASCAL, how the kernel is used from it, and the typical structure of device drivers--especially the interface to a requesting program.

The overall structure of the program is:

```
module lcdriver;
include $dap, $unibus;

   const
   .
   .
   type
   .
   .
   var
   .
   .
   program lpdriver;
      {Block.}

   interrupt_service powerfail_recovery(...);
      {Block.}

   interrupt_service printer_interrupt(...);
      {Block.}

end; {End of module.}
```

The module and end enclose a single compilation unit for the compiler, containing, in this case, a program block; constant, type, and variable declarations; and two interrupt service routines. The items declared at the module's outer level (that is, outside any block) are usable everywhere in the module. Interrupt service routines must be declared at the outer level.

Separate Compilation Allows Shared Modules.
When a module is compiled, items declared at its outer level can be "exported," allowing them to be used in other modules. Exported names, which by default are all the names declared at the outer level, are written in a special symbol table in the object module. When such an object module is included in the compilation of another module, the export symbol table is read by the compiler to get declarations as needed. The symbol table has type information for each name, so type checking is retained among separately compiled modules. The module heading also has import and export options that control the availability of names between modules.

The words $dap and $unibus are the names of other previously compiled object modules that contain declarations used in this one. They are, respectively, the declarations of Data Access Protocol (DAP) interfaces and of support routines for writing UNIBUS device drivers.

Attributes.
The type section, among other things, declares the types of several device control registers and of the communication region used by the driver and the interrupt service routines:

```
type
   byte = 0..255;
   word = 0..65535;

   csr = [word] packed record
            print_en : [pos(0)] boolean; {printer enable}
            reset : [pos(1)] boolean; {master reset}
            format : [pos(2)] boolean; {format flag}
            unused 1 : [pos(3)] boolean; {bit 4 unused}
            maint : [pos(5)] boolean; {maint bit}
            int_en : [pos(6)] boolean; {interrupt enable}
            ready : [pos(7)] boolean; {printer ready}
            irnum : 0..7;       {indirect register number}
            cverf : [pos(12)] boolean; {cable verify}
            daverr : [pos(13)] boolean; {davfu error}
            lpt_err : [pos(14)] boolean; {line printer error}
            nxm_err : [pos(15)] boolean; {nonexistent memory}
   end;     {record}

      .
      .
   register_ptr = ↑lpt_registers;

   comm_region = record
            csr_response : csr;
            bytes_output : word;
            lines_output : word;
            base_vector_address : integer;
            powerfail : boolean;
            device_busy : boolean;
         end; {record}
   region_ptr = ↑ comm_region;
   .
   .
```

The attribute [WORD] means that the entire packed record is contained in a 16-bit word. [POS] means that the field begins at the given bit offset from the record's base.

Main Program.
The program obtains the name of the lineprinter controller it is to access from its argument list. The user supplies the name with the System Builder when the driver is built into the system.

The name is then used in the CREATE_DEVICE call, which creates the DEVICE value used to synchronize the program with lineprinter interrupts (the value is returned in the printer_device parameter). As part of the operation, the kernel uses the controller name to locate the system's description of the printer device interface, created by the System Builder.

```
program lpdriver;

  [INLINE] procedure initialize_interface;
    . . . {Block.}
          .
  function open_printer of type dap$open_action;
    ...{Block.}

  function write_printer of type dap$put_action;
    ...{Block.}

begin

  {Get the name of the controller}
  controller_name := program_argument (1);

  {Create the device}
  create_device (controller_name,
        printer_device,
        vector_number   := lp_vector_number,
        vector    := vector,
        region    := region,
        registers    := register_location,
        priority    := printer_ipl,
        service_routine   := printer_interrupt,
        powerfail_routine   := powerfail_recovery);
```

A device description consists of the address of the device's first control/status register (returned in register_location), the address of the device's interrupt vector (returned in vector), and the interrupt priority level (returned in printer_ipl). The procedure creates a communication region large enough to hold a data item of type comm_region and returns its address in the region parameter. The inputs to CREATE_DEVICE are two interrupt service routines, one of which actually handles device interrupts and the other of which handles power failure occurring while the driver is running.

The program then appends "0" (by convention) to the controller name and makes it the name for its own job port. This name can then be used by other programs in the system to send output to the lineprinter by using a name like LPA0:

```
{Create a name for the job's port}

device_name := controller_name + '0';
job_port (printer_port);
create_name (job_name,
      device_name,
      printer_port,
      table := name$local);
```

The program calls a procedure to initialize the device interface and calls another procedure, INITIALIZATION_DONE, to inform the kernel that it has completed its initialization sequence. Until INITIALIZATION_DONE was called, the kernel would not start other jobs. This allows a programmer to configure a system so that all drivers are initialized before they are needed.

```
{Initialize the DMF-32 interface for the line printer}
initialize_interface;

{Signal the kernel that we're done with initialization}
initialization_done;
```

Finally, the program sits in a loop that waits for and accepts a circuit request (a message from the I/O runtime code), calls DAP$SERVER to handle the circuit, and then breaks the circuit.

```
    {Now wait for someone who wants to use the printer}
    while true do
        begin

        {Accept the circuit and go process the request}
        accept_circuit (printer_port);
        status := dap$server (circuit_port := printer_port,
                open_action := open_printer,
                put_action := write_printer);

        {Disconnect the circuit and wait for another one}
        disconnect_circuit (printer_port)
        end

end;
```

The messages sent and received on the circuit all conform to the Data Access Protocol, the interface which is defined in $DAP. In most device drivers, the function DAP$SERVER is called with two kinds of arguments:

- A message port that represents the requestor's circuit on which the $DAP module should operate
- A list of action routines, which perform each of the appropriate actions on the specific device (here, routines are supplied for "open" and "put")

DAP$SERVER applies one of the given action routines to the message and returns a status value indicating whether the action was successful.

Dap Action Routine.
The function WRITE-PRINTER is a function of type dap$put_action. Function types pre-define a function's parameters and its result. In this case, the function has six input parameters and returns a status value. The $DAP module has exported this function type.

```
function write_printer of type dap$put_action;
{+ +
{
{Put_action : put/write action routine
{
{Inputs:    record_access     -record access type
{       :   record_number     -record number
{       :   record_options    -record options
{       :   buffer            -output buffer
{       :   buffer_length     -length of data in buffer
{       :   context           -driver specific parameter (unused)
{
{Outputs:   return value      -write status
{
{--}
```

```
var
    prefix_char : char;
    prefix_count : byte;
    postfix_char : char;
    postfix_count : byte;
    char_count : integer;
    first_char : char;
```

Basically, the ''write,'' or ''put,'' action is: interpret the first character in the record, especially if the output file has ''FORTRAN_type'' carriage control; load the device registers with the appropriate information; start the transfer; and wait for the transfer to complete. A more complete explanation of this carriage control treatment can be found in the VMS documentation for terminal or lineprinter drivers.

```
begin
  {These are defaults for prefix and postfix characters and counts}
  prefix_char := chr(line_feed);
  prefix_count := 1;
  postfix_char := chr(carriage_return);
  postfix_count := 1;

  char_count := buffer_length;

  {Zero length buffer?}
  if buffer_length = 0 then
    mapped_address.long := 0
  else
```

```
    {Check for fortran-type carriage control}
    if fortran then
       begin
       first_char := substr(buffer↑, 1, 1);

       {Check for double spacing}
       if first_char = '0' then prefix_count := 2;

       {Check for formfeed}
       if first_char = '1' then prefix_char := chr(form_feed);

       {Check for overstrike}
       if first_char = '+' then prefix_count := 0;
       {Don't output first character}
       char_count := char_count - 1;
       end;


{Now start setting up the interface registers}

repeat

   region↑.powerfail := false;

   if buffer_length <> 0 then
      begin

      {Map the user's buffer (or remap it if a powerfail occurred)}
      unibus_map (dev       := printer_device,
            buffer      := buffer↑,
            buffer_size  := buffer_length,
            unibus_address := mapped_address.long);

      {Don't output first character for fortran carriage control}
      if fortran then
         mapped_address.long := mapped_address.long + 1;
      end;
```

```
{Synchronize with device}
disable_interrupt (printer_ipl);
```

The procedure WRITE_REGISTER lets you specify a record representing a device register (the first argument), followed by assignments to the record's fields (the second and subsequent arguments). Any fields not specified are zeroed. For all fields that are specified, the procedure constructs an output value of the appropriate length and format and writes it into the register with a single instruction.

```
{Set Indirect register number in csr}
write_register (register_location↑.control, format := true, irnum := 2);

{Write out to the postfix and prefix indirect registers (2 & 3)}
write_register (register_location↑.prefix,
            chr_cnt := prefix_count,
            ps_chr := prefix_char);
write_register (register_location↑.suffix,
            chr_cnt := postfix_count,
            ps_chr := postfix_char);
{Write out the low order 16 bits of the mapped address}
write_register (register_location ↑ .dma_low, mapped_address.low16);

{Write out the DMA character count}
write_register (register_location ↑ .dma_count, -char_count);

{Write out indirect register 6 which contains several various things}
write_register (register_location ↑ .dma_h_char,
        dma_high := mapped_address.high2,
        auto_cr := default_auto_cr,
```

```
        ff_lf := default_ff_lf,
        npc_acc := default_npc_acc,
        wrap  := default_wrap,
        lc_uc := default_lc_uc);

{Register 7 contains info about paper size}
write_register (register_location↑.lp_format,
            p_height := lines_per_page,
            p_width  := page_width);

{Raise to powerfail ipl for powerfail check}

disable_interrupt (ipl$_power);
if not region↑.powerfail then
   begin

   {Now set the device enable bit to start the transfer}
   {Reading csr and setting appropriate bits preserves csr contents}

   lpt_csr := read_register (register_location↑.control);
   lpt_csr.print_en := true;
   lpt_csr.int_en := true;
   write_register (register_location↑.control, lpt_csr);
   region↑.device_busy := true;
   enable_interrupt;
```

The WAIT_ANY procedure will put the driver process in the waiting state until the interrupt service routine calls SIGNAL_DEVICE.

```
   {Now we wait for the transfer to complete}
   wait_any (printer_device);
   end
```

```
        else
            enable_interrupt;
        if buffer_length <> 0 then
            {Unmap the buffer (even if powerfail ... it gets remapped later)}
            unibus_unmap (dev      := printer_device,
                  buffer       := buffer↑,
                  buffer_size  := buffer_length,
                  unibus_address := mapped_address::integer);
    until not region↑.powerfail;
    {Indicate success or failure}
    if region↑.csr_response.lpt_err then
        write_printer := dap$k_device_error
    else
        write_printer := dap$k_success
end; {End of action routine.}
```

When the controller finishes with the request, an interrupt occurs. The interrupt service routine reads the contents of the device register into the communication region so that the main driver program can access the information. It then calls SIGNAL_DEVICE so that the main program can proceed with the WRITE_PRINTER action.

```
interrupt_service printer_interrupt (xreg : register_ptr;
                                     comm : region_ptr);
begin
    {Read the CSR and indirect registers 0 and 1}
    with comm↑, x reg↑ do
        begin
        csr_response := read_register (control);
        bytes_output := read_register (bytes_out);
        lines_output := read_register (lines_out);
        device_busy := false;
        end;
    signal_device
end;
```

Another interrupt service routine is called by the kernel when a power failure interrupt occurs. It performs a controlled reinitialization of the lineprinter interface.

```
interrupt_service powerfail_recovery (xregister : register_ptr;
                                      comm    : region_ptr);
var
   combo_board_csr0 : combo_csr0;
begin
   with comm ↑, xregister ↑ do
      begin

   {Reset the interface}
   write_register (control, reset : = true);
   .
   .
   powerfail : = true;
   if device_busy then
   signal_device;
end; {with}
end;
```

A

ACCEPT_CIRCUIT procedure, 37, 62
ALLOCATE_MEMORY procedure, 63
ANYTYPE data type, 25
applications
 developed under VAXELN, 28-33
 of VAXELN programs, 2
architecture, of VAX systems, 41-42
attributes, 73-74

B

baseband signaling, 39
Basic Agreements, 53
Basic Service, 54
binary semaphores, 15
BOOLEAN function type, 25
BYTE_DATA type, 24, 25

C

circuits, 17, 37-38
CLEAR_EVENT procedure, 13, 63
COBOL, 41
code optimization, 30
communications
 circuits for, 37-38
 distributed processing networking for, 38-39
 Ethernet for, 36-37, 39
 in kernel, 8, 17-21
 networks for, 35-37
 synchronization and, for concurrency, 12-16
compilers
 VAXELN, 31
 VAXELN PASCAL, 30
concurrent processes, 12-16
configuration, 32
CONNECT_CIRCUIT procedure, 37, 63
control region of memory, 11
CONVERT function (in VAXELN PASCAL), 25
CREATE_DEVICE procedure, 64
CREATE_EVENT procedure, 64
CREATE_JOB procedure, 9, 65
CREATE_MESSAGE procedure, 19, 65
CREATE_NAME procedure, 65
CREATE_PORT procedure, 65
CREATE procedures, 9, 17
CREATE_PROCESS procedure, 9, 66
 PROCESS_BLOCK procedures activated by, 25
CREATE_SEMAPHORE procedure, 66
 CSMA/CD access control, 39
CURRENT_PROCESS procedure, 66

Customer Runnable Diagnostics (CRDS), 54

D

dap action routine, 78-84
Data Access Protocol (DAP), 26
database management, VAX Rdb/ELN for, 6, 56-61
datagrams (messages), 36
data types, 23-26
debuggers, 5, 32-33
DECnet, 36-37
DECnet-VAX Network Control Program (NCP), 36
DECservice, 54
DECsupport, 53
DELETE procedure, 9, 66
demand paging, 11
DEVICE data type, 25
device drivers, 8, 16
 lineprinter, 72-84
device objects, 16
DEVICE values, 16
diagnostics, 54
Digest, 55
Digital Standard Relational Interface (DSRI), 57, 58
DISABLE_ASYNCH_EXCEPTION procedure, 67
DISABLE_SWITCH procedure, 67
DISCONNECT_CIRCUIT procedure, 67
distributed processing, 29, 35-36, 38-39
driver, lineprinter, 72-84

E

EDIT mode (System Builder), 31
EDT (text editor), 29-30
Education Services, 55
ENABLE_ASYNCH_EXCEPTION procedure, 67
ENABLE_SWITCH procedure, 67
Ethernet, 4, 35-37
 VAX Rdb/ELN used with, 56
EVENT data type, 25
EVENT objects, 13
events, 13-14
EXCEPTION_HANDLER function type, 24, 25
EXIT procedure, 67
export symbol table, 30

F

File Service, 26, 28
FIND procedure, 26
FORTRAN, 41
FREE_MEMORY procedure, 68
functions, in VAXELN PASCAL, 23, 24
FUNCTION_TYPE directive, 23
function types, in VAXELN PASCAL, 25

G

GET_TIME procedure, 68

H

hardware
 kernel as layer between software and, 8
 MicroVAX I, 42-43
 services for, 54
 VAX-11/725, 43-44
 VAX-11/730, 44
 VAX-11/750, 45-46
 VAX-11/780, 46-47
 VAX-11/782, 48
 VAX-11/785, 49
 VAX clusters, 49-50
 of VAX systems, 41-42
HELP facilities, in EDT, 30
host processors, 3

I

INITIALIZATION_DONE procedure, 68
INLINE attribute, 24
instruction set, 41
International Standards Organization (ISO), Pascal standard of, 23
interrupts, 16
INTERRUPT_SERVICE procedure, 16, 24, 25
interrupt service routine (ISR), 16
 in VAXELN PASCAL, 25
I/O operations, 11
 on networks, 35
 VAXELN File Service for, 26

J

JOB_PORT procedure, 68
jobs (in processes), 9-10
 messages passed between, 17
journaling, in EDT, 29

K

KER$ALLOCATE_MAP procedure, 62
KER$ALLOCATE_PATH procedure, 62
KER$FREE_MAP procedure, 67-68
KER$FREE_PATH procedure, 68
kernel, 8-11
 communications in, 17-21
 communications and synchronization in, 12-16
 multitasking and multiprogramming supported by, 5
 procedures, 62-71
kernel data types, 25
kernel objects, 8-9

L

languages
 available on VAX/VMS operating system, 41
 VAXELN PASCAL, 23-26

LCDRIVER (lineprinter driver), 72-84
libraries, 30-31
lineprinter driver, 72-84
Linker, 29, 31
local area networks, 4, 35, 38
 VAX Rdb/ELN used on, 56
local names, 13
LOCATE procedure, 26
Loopback Mirror, 37

M

MA780 Multiport Memory option, 47, 48
mapping, 11
memory
 on MicroVAX I, 42
 on VAX-11/725, 43-44
 on VAX-11/730, 44
 on VAX-11/750, 45
 on VAX-11/780, 46, 47
 on VAX-11/782, 48
memory management, used by VAXELN, 11
menus, in System Builder, 31
MESSAGE data type, 25
messages, 13
 communications of, 17-21
 ports for queues of, 12
 transmitted along circuits, 37-38
 transmitted between Ethernet nodes, 35, 36
 transmitted by Network Service, 37
MicroVAX I, 42-43
 VAX Rdb/ELN used on, 56, 57
MicroVMS operating system, VAXELN Toolkit option with, 3, 28
modules
 compiling of, 30
 shared, 73
multiprogramming, 5
multitasking, 5
MUTEX data type, 15, 26

N

NAME data type, 25
Names, for ports, 12-13
name servers, 35
Network Control Program (NCP), 36
Network Management Listener (NML), 36
networks, distributed processing on, 35-36, 38-39
Network Service, 4, 28, 29, 35
 messages passed by, 18, 37
 messages segmented by, 36
 names translated by, 13
Network Service Protocol (NSP), 36

O

object modules, 29
objects, 17

device, 16
kernel, 8-9
optimization of code, 30

P

page-faulting, 11
parameters, in VAXELN PASCAL, 24
PASCAL, *see* VAXELN PASCAL
PDP-11 systems, 42
Phase IV DECnet Network Service Protocol (NSP), 36
pointers, for messages, 19
PORT data type, 12, 25
ports, 12
circuits and, 17
printer driver, 72-84
priorities, for jobs on same processor, 10-11
procedure calls, 8
procedures
kernel, 62-71
in VAXELN PASCAL, 24
PROCEDURE_TYPE directive, 23
procedure types, in VAXELN PASCAL, 25
PROCESS_BLOCK procedure, 25
process blocks, 9, 25
PROCESS data type, 25
processes, 9-11
concurrent, 12
processors
MicroVAX, 42-43
priorities for jobs on, 10
VAX-11/725, 43-44
VAX-11/730, 44
VAX-11/750, 45-46
VAX-11/780, 46-47
VAX-11/782, 48
VAX-11/785, 49
Programmed Array Logic (PAL), 44
programming
using VAXELN, 4
in VAXELN PASCAL, 23-26
program region of memory, 11
programs
in VAXELN, 3-4
VAXELN applications, 28
protocols
DECnet, 36
for messages, 21

Q

Q-bus, 42
QUEUE_ENTRY data type, 26

R

RAISE_EXCEPTION procedure, 69
read only memory (ROM), applications put on, 3
realtime applications
VAXELN programs for, 2

VAX Rdb/ELN used for, 56-57
RECEIVE procedure, 20, 69
Record Management Service (RMS), 26
relational databases, VAX Rdb/ELN for, 56-61
Remote Diagnosis (RD), 54
RESUME procedure, 10, 69

S

scheduling, of jobs on processors, 10
SEMAPHORE data type, 25
semaphores, 14-15
SEND procedure, 19, 35, 69
server networks, 38-39
service
 education services, 55
 hardware, 54
 software, 52-53
Session Control Protocol, 36
SET_JOB_PRIORITY procedure, 69-70
SET_PROCESS_PRIORITY procedure, 70
SET_TIME procedure, 70
shared modules, 73
SIGNAL_DEVICE procedure, 16, 70, 82, 83
SIGNAL procedure, 10, 13-15, 70
software
 kernel as layer between hardware and, 8
 service for, 52-53
Software Infomation Network, 53
statically defined systems, 2
STRING data type, 24
support, 52
SUSPEND procedure, 10, 70
swapping, 11
symbolic debugger, 5, 32-33
symbol tables, 30
System Builder utility, 3, 29, 31, 32
 Network Service included in, 35
System Building Block (SBB), 41-42

T

target processors, 3
tasks, processes as, 9
text editor (EDT), 29-30
TRANSLATE_NAME procedure, 12, 70-71
typecasting, 25

U

UNIBUS, 43
universal names, 13
User Control Store, 46
user programs, 28
user stack, 11

utilities
for application development, 29
System Builder, 31, 32
VAXELN Toolkit used with, 3

V

variables, binary semaphores to guard, 15
VARYING_STRING data type, 24
VAX-11 COBOL, 41
VAX-11 FORTRAN, 41
VAXclusters, 49-50
VAXELN, 2-5
applications developed under, 28-33
communications under, 35-39
concurrent processes in, 12-16
processes in, 9-11
VAX Rdb/ELN and Ethernet used with, 56
VAXELN File Service, 26
VAXELN host processor, 3
VAXELN kernel, *see* kernel
VAXELN Network Management Listener (NML), 36
VAXELN Network Service, *see* Network Service
VAXELN PASCAL, 3, 4, 23-26
application programs written in, 28-29
compiler, 30
device drivers in, 16
multitasking in, 5
PORT data type in, 12
VAXELN Toolkit, 3, 28
VAX Rdb/ELN, 6, 56-61
VAX Rdb/VMS, 56, 57
VAX systems, 41-42
hardware services for, 54
MicroVAX I, 42-43
VAX-11/725, 43-44
VAX-11/730, 44
VAX-11/750, 45-46
VAX-11/780, 46-47
VAX-11/782, 48
VAX-11/785, 49
VAX clusters, 49-50
VAX/VMS Librarian, 31
VAX/VMS Linker, 29, 31
VAX/VMS operating system, 33, 41
VAXELN Toolkit option with, 3, 28

W

WAIT_ALL procedure, 10, 71
WAIT_ANY procedure, 10, 71, 82
WAIT procedures, 14
WRITE_PRINTER procedure, 83
WRITE_REGISTER procedure, 81

Notes

digital